WEST

Generate Collection | Print

(5)

L12: Entry 2 of 3 File: USPT Apr 22, 2003

DOCUMENT-IDENTIFIER: US 6553408 B1
TITLE: Virtual device architecture having memory for storing lists of driver modules

Detailed Description Text (40):
The ISAN server 102A has four separate 64-bit 66 MHz PCI busses 200A-D. Many different configurations of storage devices and network interfaces in the slots of the PCI busses are possible. In one embodiment, the PCI busses are divided into two groups: the SSD PCI busses 200A-B and the interface PCI busses 200C-D. Each group has two busses that are designated by the terms upper and lower. The upper and lower busses in each group can be configured to provide redundant services. For example, the lower SSD PCI bus 200B has the same configuration as the upper SSD PCI bus 200A.

Detailed Description Text (79):
FIG. 10 illustrates a partition ISM 750. The partition ISM 750 includes an interface 751 which receives internal communications from other driver modules, and an interface 752 which also communicates with other driver modules. The ISM 750 includes logic processes 753, data structures for storing a base address 754 and a limit address 755, and a drive interface 756. The partition logic process 753 configures the subject storage device identified by the drive process 756, using a logical partitioning function useful for a variety of storage management techniques, so that the physical device appears as more than one logical device in the virtual circuits. In this example, the partition ISM 750 is an instance of a class "partition," and given device identifier 10400.

Detailed Description Text (106):
The connection option such as the network interface 146 over which the storage transaction is received is coupled to a hardware device driver. The hardware device driver receives the storage transaction and depending on the protocol, dispatches it to an appropriate virtual device for handling that protocol.

(2)

WEST

Generate Collection Print

L9: Entry 2 of 13 File: USPT Oct 14, 2003

DOCUMENT-IDENTIFIER: US 6633916 B2
TITLE: Method and apparatus for virtual resource handling in a multi-processor computer system

Abstract Text (1):
Multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources is subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system. At different times, different operating system instances may be loaded on a given partition. Resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration. The partitions themselves can also be changed without rebooting the system by modifying the configuration tree. Each instance keeps track of the CPUs in the system and their respective operational statuses relative to the instance, such as compatibility with the instance, control by the instance, and availability to the instance for SMP processing. Using separate bitvectors for the different categories, a single bit in each bitvector may be used to represent the membership of a given CPU in that particular set, and therefore in the category represented by that set.

Brief Summary Text (2):
This invention relates to multiprocessor computer architectures in which processors and other computer hardware resources are grouped in partitions, each of which has an operating system instance and, more specifically, to methods and apparatus for identifying different processing resources with the operating system instances.

Brief Summary Text (5):
Traditionally, computing speed has been addressed by using a "shared nothing" computing architecture where data, business logic, and graphic user interfaces are distinct tiers and have specific computing resources dedicated to each tier. Initially, a single central processing unit was used and the power and speed of such a computing system was increased by increasing the clock rate of the single central processing unit. More recently, computing systems have been developed which use several processors working as a team instead one massive processor working alone. In this manner, a complex application can be distributed among many processors instead of waiting to be executed by a single processor. Such systems typically consist of several central processing units (CPUs) which are controlled by a single operating system. In a variant of a multiple processor system called "symmetric multiprocessing" or SMP, the applications are distributed equally across all processors. The processors also share memory. In another variant called "asymmetric multiprocessing" or AMP, one processor acts as a "master" and all of the other processors act as "slaves." Therefore, all operations, including the operating system, must pass through the master before being passed onto the slave processors. These multiprocessing architectures have the advantage that performance can be increased by adding additional processors, but suffer from the disadvantage that the software running on such systems must be carefully written to take advantage of the multiple processors and it is difficult to scale the software as the number of processors increases. Current commercial workloads do not scale well beyond 8-24 CPUs as a single SMP system, the exact number depending upon platform, operating system and application mix.

Brief Summary Text (8):
The availability and maintainability issues were addressed by a "shared everything" model in which a large centralized robust server that contains most of the resources is networked with and services many small, uncomplicated client network computers. Alternatively, "clusters" are used in which each system or "node" has its own memory and is controlled by its own operating system. The systems interact by sharing disks and passing messages among themselves via some type of communications network. A cluster system has the advantage that additional systems can easily be added to a cluster. However, networks and clusters suffer from a lack of shared memory and from limited interconnect bandwidth which places limitations on performance.

Brief Summary Text (9):
In many enterprise computing environments, it is clear that the two separate computing models must be simultaneously accommodated and each model optimized. Several prior art approaches have been used to attempt this accommodation. For example, a design called a "virtual machine" or VM developed and marketed by International Business Machines Corporation, Armonk, N.Y., uses a single physical machine, with one or more physical processors, in combination with software which simulates multiple virtual machines. Each of those virtual machines has, in principle, access to all the physical resources of the underlying real computer. The assignment of resources to each virtual machine is controlled by a program called a "hypervisor". There is only one hypervisor in the system and it is responsible for all the physical resources. Consequently, the hypervisor, not the other operating systems, deals with the allocation of physical hardware. The hypervisor intercepts requests for resources from the other operating systems and deals with the requests in a globally-correct way.

Brief Summary Text (11):
In addition, the CPUs logically assigned to each partition can be turned "on" and "off" dynamically via normal operating system operator commands without re-boot. The only limitation is that the number of CPUs active at system intitialization is the maximum number of CPUs that can be turned "on" in any partition.

Brief Summary Text (15):
The Hive Project conducted at Stanford University uses an architecture which is structured as a set of cells. When the system boots, each cell is assigned a range of nodes that it owns throughout execution. Each cell manages the processors, memory and I/O devices on those nodes as if it were an independent operating system. The cells cooperate to present the illusion of a single system to user-level processes.

Brief Summary Text (17):
A system called "Cellular IRIX" developed and marketed by Silicon Graphics Inc. Mountain View, Calif., supports modular computing by extending traditional symmetric multiprocessing systems. The Cellular IRIX architecture distributes global kernel text and data into optimized SMP-sized chunks or "cells". Cells represent a control domain consisting of one or more machine modules, where each module consists of processors, memory, and I/O. Applications running on these cells rely extensively on a full set of local operating system services, including local copies of operating system text and kernel data structures. Only one instance of the operating system exists on the entire system. Inter-cell coordination allows application images to directly and transparently utilize processing, memory and I/O resources from other cells without incurring the overhead of data copies or extra context switches.

Brief Summary Text (18):
Another existing architecture called NUMA-Q developed and marketed by Sequent Computer Systems, Inc., Beaverton, Oreg. uses "quads", or a group of four processors per portion of memory, as the basic building block for NUMA-Q SMP nodes. Adding I/O to each quad further improves performance. Therefore, the NUMA-Q architecture not only distributes physical memory but puts a predetermined number of processors and PCI slots next to each part. The memory in each quad is not local memory in the traditional sense. Rather, it is one third of the physical memory address space and has a specific address range. The address map is divided evenly over memory, with each quad containing a contiguous portion of address space. Only one copy of the operating system is running and, as in any SMP system, it resides in memory and runs processes without distinction and simultaneously on one or more processors.

Brief Summary Text (19):
Accordingly, while many attempts have been made at providing a flexible computer system having maximum resource availability and scalability, existing systems each have significant shortcomings. Therefore, it would be desirable to have a new computer system design which provides improved flexibility, resource availability and scalability. Furthermore, to allow the proper handling of a plurality of resources in a multiple processor environment, it would be desirable to provide some framework by which they could be identified by an operating system and by which they could appropriately be applied.

Brief Summary Text (21):
In accordance with the principles of the present invention, multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources is adaptively subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system. Each of the partitions has access to its own physical resources plus resources designated as shared. In accordance with one embodiment, the partitioning of resources is performed by assigning resources within a configuration.

Brief Summary Text (22):
More particularly, software logically, and adaptively, partitions CPUs, memory, and I/O ports by assigning them together. An instance of an operating system may then be loaded on a partition. At different times, different operating system instances may be loaded on a given partition. This partitioning, which a system manager directs, is a software function; no hardware boundaries are required. Each individual instance has the resources it needs to execute independently. Resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration. The partitions themselves can also be changed without rebooting the system by modifying the configuration tree. The resulting adaptively-partitioned, multi-processing (APMP) system exhibits both scalability and high performance.

Drawing Description Text (10):
FIGS. 8A and 8B, when placed together, form a flowchart illustrating the steps in an illustrative routine followed by an operating system instance to join an APMP computer system which is already created.

Detailed Description Text (2):
A computer platform constructed in accordance with the principles of the present invention is a multi-processor system capable of being partitioned to allow the concurrent execution of multiple instances of operating system software. The system does not require hardware support for the partitioning of its memory, CPUs and I/O subsystems, but some hardware may be used to provide additional hardware assistance for isolating faults, and minimizing the cost of software engineering. The following specification describes the interfaces and data structures required to support the inventive software architecture. The interfaces and data structures described are not meant to imply a specific operating system must be used, or that only a single type of operating system will execute concurrently. Any operating system which implements the software requirements discussed below can participate in the inventive system operation.

Detailed Description Text (6):
An APMP computer system 200 constructed in accordance with the principles of the present invention from a software view is illustrated in FIG. 2. In this system, the hardware components have been allocated to allow concurrent execution of multiple operating system instances 208, 210, 212.

Detailed Description Text (7):
In a preferred embodiment, this allocation is performed by a software program called a "console" program, which, as will hereinafter be described in detail, is loaded into memory at power up. Console programs are shown schematically in FIG. 2 as programs 213, 215 and 217. The console program may be a modification of an existing

administrative program or a separate program which interacts with an operating system to control the operation of the preferred embodiment. The console program does not virtualize the system resources, that is, it does not create any software layers between the running operating systems 208, 210 and 212 and the physical hardware, such as memory and I/O units (not shown in FIG. 2.) Nor is the state of the running operating systems 208, 210 and 212 swapped to provide access to the same hardware. Instead, the inventive system logically divides the hardware into partitions. It is the responsibility of operating system instance 208, 210, and 212 to use the resources appropriately and provide coordination of resource allocation and sharing. The hardware platform may optionally provide hardware assistance for the division of resources, and may provide fault barriers to minimize the ability of an operating system to corrupt memory, or affect devices controlled by another operating system copy.

Detailed Description Text (8):
The execution environment for a single copy of an operating system, such as copy 208 is called a "partition" 202, and the executing operating system 208 in partition 202 is called "instance" 208. Each operating system instance is capable of booting and running independently of all other operating system instances in the computer system, and can cooperatively take part in sharing resources between operating system instances as described below.

Detailed Description Text (9):
In order to run an operating system instance, a partition must include a hardware restart parameter block (HWRPB), a copy of a console program, some amount of memory, one or more CPUs, and at least one I/O bus which must have a dedicated physical port for the console. The HWRPB is a configuration block which is passed between the console program and the operating system.

Detailed Description Text (10):
Each of console programs 213, 215 and 217, is connected to a console port, shown as ports 214, 216 and 218, respectively. Console ports, such as ports 214, 216 and 218, generally come in the form of a serial line port, or attached graphics, keyboard and mouse options. For the purposes of the inventive computer system, the capability of supporting a dedicated graphics port and associated input devices is not required, although a specific operating system may require it. The base assumption is that a serial port is sufficient for each partition. While a separate terminal, or independent graphics console, could be used to display information generated by each console, preferably the serial lines 220, 222 and 224, can all be connected to a single multiplexer 226 attached to a workstation, PC, or LAT 228 for display of console information.

Detailed Description Text (12):
Partitions can be "initialized" or "uninitialized." An initialized partition has sufficient resources to execute an operating system instance, has a console program image loaded, and a primary CPU available and executing. An initialized partition may be under control of a console program, or may be executing an operating system instance. In an initialized state, a partition has full ownership and control of hardware components assigned to it and only the partition itself may release its components.

Detailed Description Text (14):
An uninitialized partition is a partition which has no primary CPU executing either under control of a console program or an operating system. For example, a partition may be uninitialized due to a lack of sufficient resources at power up to run a primary CPU, or when a system administrator is reconfiguring the computer system. When in an uninitialized state, a partition may reassign its hardware components and may be deleted by another partition. Unassigned resources may be assigned by any partition.

Detailed Description Text (15):
Partitions may be organized into "communities" which provide the basis for grouping separate execution contexts to allow cooperative resource sharing. Partitions in the same community can share resources. Partitions that are not within the same community cannot share resources. Resources may only be manually moved between

partitions that are not in the same community by the system administrator by deassigning the resource (and stopping usage), and manually reconfiguring the resource. Communities can be used to create independent operating system domains, or to implement user policy for hardware usage. In FIG. 2, partitions 202 and 204 have been organized into community 230. Partition 206 may be in its own community 205. Communities can be constructed using the configuration tree described below and may be enforced by hardware.

Detailed Description Text (18):
More specifically, when the APMP system is powered up, a CPU will be selected as a primary CPU in a conventional manner by hardware which is specific to the platform on which the system is running. The primary CPU then loads a copy of a console program into memory. This console copy is called a "master console" program. The primary CPU initially operates under control of the master console program to perform testing and checking assuming that there is a single system which owns the entire machine. Subsequently, a set of environment variables are loaded which define the system partitions. Finally, the master console creates and initializes the partitions based on the environment variables. In this latter process the master console operates to create the configuration tree, to create additional HWRPB data blocks, to load the additional console program copies, and to start the CPUs on the alternate HWRPBs. Each partition then has an operating system instance running on it, which instance cooperates with a console program copy also running in that partition. In an unconfigured APMP system, the master console program will initially create a single partition containing the primary CPU, a minimum amount of memory, and a physical system administrator's console selected in a platform-specific way. Console program commands will then allow the system administrator to create additional partitions, and configure I/O buses, memory, and CPUs for each partition.

Detailed Description Text (19):
After associations of resources to partitions have been made by the console program, the associations are stored in non-volatile RAM to allow for an automatic configuration of the system during subsequent boots. During subsequent boots, the master console program must validate the current configuration with the stored configuration to handle the removal and addition of new components. Newly-added components are placed into an unassigned state, until they are assigned by the system administrator. If the removal of a hardware component results in a partition with insufficient resources to run an operating system, resources will continue to be assigned to the partition, but it will be incapable of running an operating system instance until additional new resources are allocated to it.

Detailed Description Text (20):
As previously mentioned, the console program communicates with an operating system instance by means of an HWRPB which is passed to the operating system during operating system boot up. The fundamental requirements for a console program are that it should be able to create multiple copies of HWRPBs and itself. Each HWRPB copy created by the console program will be capable of booting an independent operating system instance into a private section of memory and each operating system instance booted in this manner can be identified by a unique value placed into the HWRPB. The value indicates the partition, and is also used as the operating system instance ID.

Detailed Description Text (21):
In addition, the console program is configured to provide a mechanism to remove a CPU from the available CPUs within a partition in response to a request by an operating system running in that partition. Each operating system instance must be able to shutdown, halt, or otherwise crash in a manner that control is passed to the console program. Conversely, each operating system instance must be able to reboot into an operational mode, independently of any other operating system instance.

Detailed Description Text (22):
Each HWRPB which is created by a console program will contain a CPU slot-specific database for each CPU that is in the system, or that can be added to the system without powering the entire system down. Each CPU that is physically present will be marked "present", but only CPUs that will initially execute in a specific partition

will be marked "available" in the HWRPB for the partition. The operating system instance running on a partition will be capable of recognizing that a CPU may be available at some future time by a present (PP) bit in a per-CPU state flag fields of the HWRPB, and can build data structures to reflect this. When set, the available (PA) bit in the per-CPU state flag fields indicates that the associated CPU is currently associated with the partition, and can be invited to join SMP operation.

Detailed Description Text (24):
As previously mentioned, the master console program creates a configuration tree which represents the hardware configuration, and the assignment of each component in the system to each partition. Each console program then identifies the configuration tree to its associated operating system instance by placing a pointer to the tree in the HWRPB.

Detailed Description Text (26):
The master console may generate a single copy of the tree which copy is shared by all operating system instances, or it may replicate the tree for each instance. A single copy of the tree has the disadvantage that it can create a single point of failure in systems with independent memories. However, platforms that generate multiple tree copies require the console programs to be capable of keeping changes to the tree synchronized.

Detailed Description Text (27):
The configuration tree comprises multiple nodes including root nodes, child nodes and sibling nodes. Each node is formed of a fixed header and a variable length extension for overlaid data structures. The tree starts with a tree root node 302 representing the entire system box, followed by branches that describe the hardware configuration (hardware root node 304), the software configuration (software root node 306), and the minimum partition requirements (template root node 308.) In FIG. 3, the arrows represent child and sibling relationships. The children of a node represent component parts of the hardware or software configuration. Siblings represent peers of a component that may not be related except by having the same parent. Nodes in the tree 300 contain information on the software communities and operating system instances, hardware configuration, configuration constraints, performance boundaries and hot-swap capabilities. The nodes also provide the relationship of hardware to software ownership, or the sharing of a hardware component.

Detailed Description Text (28):
The nodes are stored contiguously in memory and the address offset from the tree root node 302 of the tree 300 to a specific node forms a "handle" which may be used from any operating system instance to unambiguously identify the same component on any operating system instance. In addition, each component in the inventive computer system has a separate ID. This may illustratively be a 64-bit unsigned value. The ID must specify a unique component when combined with the type and subtype values of the component. That is, for a given type of component, the ID must identify a specific component. The ID may be a simple number, for example the CPU ID, it may be some other unique encoding, or a physical address. The component ID and handle allow any member of the computer system to identify a specific piece of hardware or software. That is, any partition using either method of specification must be able to use the same specification, and obtain the same result.

Detailed Description Text (29):
As described above, the inventive computer system is composed of one or more communities which, in turn, are composed of one or more partitions. By dividing the partitions across the independent communities, the inventive computer system can be placed into a configuration in which sharing of devices and memory can be limited. Communities and partitions will have IDs which are densely packed. The hardware platform will determine the maximum number of partitions based on the hardware that is present in the system, as well as having a platform maximum limit. Partition and community IDs will never exceed this value during runtime. IDs will be reused for deleted partitions and communities. The maximum number of communities is the same as the maximum number of partitions. In addition, each operating system instance is identified by a unique instance identifier, for example a combination of the partition ID plus an incarnation number.

Detailed Description Text (32):
Hardware components place configuration constraints on how ownership may be divided. A "config" handle in the configuration tree node associated with each component determines if the component is free to be associated anywhere in the computer system by pointing to the hardware root node 304. However, some hardware components may be bound to an ancestor node and must be configured as part of this node. Examples of this are CPUs, which may have no constraints on where they execute, but which are a component part of a system building block (SBB), such as SBBs 322 or 324. In this case, even though the CPU is a child of the SBB, its config handle will point to the hardware root node 304. An I/O bus, however, may not be able to be owned by a partition other than the partition that owns its I/O processor. In this case, the configuration tree node representing the I/O bus would have a config handle pointing to the I/O processor. Because the rules governing hardware configuration are platform specific, this information is provided to the operating system instances by the config handle.

Detailed Description Text (36):
The configuration tree 300 may extend to the level of device controllers, which will allow the operating system to build bus and device configuration tables without probing the buses. However, the tree may also end at any level, if all components below it cannot be configured independently. System software will still be required to probe for bus and device information not provided by the tree.

Detailed Description Text (38):
The minimal component requirements for a partition are provided by the information contained in the template root node 308. The template root node 308 contains nodes, 316, 318 and 320, representing the hardware components that must be provided to create a partition capable of execution of a console program and an operating system instance. Configuration editors can use this information as the basis to determine what types, and how many resources must be available to form a new partition.

Detailed Description Text (39):
During the construction of a new partition, the template subtree will be "walked", and, for each node in the template subtree, there must be a node with the same type and subtype owned by the new partition so that it will be capable of loading a console program and booting an operating system instance. If there are more than one node of the same type and subtype in the template tree, there must also be multiple nodes in the new partition. The console program will use the template to validate that a new partition has the minimum requirements prior to attempting to load a console program and initialize operation.

Detailed Description Text (41):
The total size in bytes of the memory allocated for the configuration tree is located in the first quadword of the header. The size is guaranteed to be in multiples of the hardware page size. The second quadword of the header is reserved for a checksum. In order to examine the configuration tree, an operating system instance maps the tree into its local address space. Because an operating system instance may map this memory with read access allowed for all applications, some provision must be made to prevent a non-privileged application from gaining access to console data to which it should not have access. Access may be restricted by appropriately allocating memory. For example, the memory may be page aligned and allocated in whole pages. Normally, an operating system instance will map the first page of the configuration tree, obtain the tree size, and then remap the memory allocated for configuration tree usage. The total size may include additional memory used by the console for dynamic changes to the tree.

Detailed Description Text (59):
This field specifies the minimum memory in bytes (including console_req) needed for the base memory segment for a partition. This is where the console, console structures, and operating system will be loaded for a partition. It must be greater or equal to minAlloc and a multiple of minAlloc.

Detailed Description Text (71):
This field stores the partition ID of the first operating system instance.

Detailed Description Text (98):
A community provides the basis for the sharing of resources between partitions. While a hardware component may be assigned to any partition in a community, the actual sharing of a device, such as memory, occurs only within a community. The community node 310 contains a pointer to a control section, called an APMP database, which allows the operating system instances to control access and membership in the community for the purpose of sharing memory and communications between instances. The APMP database and the creation of communities are discussed in detail below. The configuration ID for the community is a signed 16-bit integer value assigned by the console program. The ID value will never be greater than the maximum number of partitions that can be created on the platform.

Detailed Description Text (99):
A partition node, such as node 312 or 314, represents a collection of hardware that is capable of running an independent copy of the console program, and an independent copy of an operating system. The configuration ID for this node is a signed 16-bit integer value assigned by the console. The ID will never be greater than the maximum number of partitions that can be created on the platform. The node has the definition:

Detailed Description Text (108):
This field holds a value which indicates the type of operating system that will be loaded in the partition.

Detailed Description Text (114):
This field holds a formatted string which is interpreted using the instance_name_format field. The value in this field provides a high-level path name to the operating system instance executing in the partition. This field is loaded by system software and is not saved across power cycles. The field is cleared at power up and at partition creation and deletion.

Detailed Description Text (118):
A memory controller node (such as nodes 336 or 350) is used to express a physical hardware component, and its owner is typically the partition which will handle errors, and initialization. Memory controllers cannot be assigned to communities, as they require a specific operating system instance for initialization, testing and errors. However, a memory description, defined by a memory descriptor node, may be split into "fragments" to allow different partitions or communities to own specific memory ranges within the memory descriptor. Memory is unlike other hardware resources in that it may be shared concurrently, or broken into "private" areas. Each memory descriptor node contains a list of subset ranges that allow the memory to be divided among partitions, as well as shared between partitions (owned by a community). A memory descriptor node (such as nodes 338 or 352) is defined as:

Detailed Description Text (123):
Fragments can have minimum allocation sizes and alignments provided in the tree root node 302. The base memory for a partition (the fragments where the console and operating system will be loaded) may have a greater allocation and alignment than other fragments (see the tree root node definition above). If the owner field of the memory descriptor node is a partition, then the fragments can only be owned by that partition.

Detailed Description Text (124):
FIG. 4 illustrates the configuration tree shown in FIG. 3 when it is viewed from a perspective of ownership. The console program for a partition relinquishes ownership and control of the partition resources to the operating system instance running in that partition when the primary CPU for that partition starts execution. The concept of "ownership" determines how the hardware resources and CPUs are assigned to software partitions and communities. The configuration tree has ownership pointers illustrated in FIG. 4 which determine the mapping of hardware devices to software such as partitions (exclusive access) and communities (shared access). An operating system instance uses the information in the configuration tree to determine to which hardware resources it has access and reconfiguration control.

Detailed Description Text (125):
Passive hardware resources which have no owner are unavailable for use until ownership is established. Once ownership is established by altering the configuration tree, the operating system instances may begin using the resources. When an instance makes an initial request, ownership can be changed by causing the owning operating system to stop using a resource or by a console program taking action to stop using a resource in a partition where no operating system instance is executing. The configuration tree is then altered to transfer ownership of the resource to another operating system instance. The action required to cause an operating system to stop using a hardware resource is operating system specific, and may require a reboot of the operating system instances affected by the change.

Detailed Description Text (128):
When a resource is de-assigned, the owner may decide to deassign the owner field, or both the owner and current_owner fields. The decision is based on the ability of the owning operating system instance running in the partition to discontinue the use of the resource prior to de-assigning ownership. In the case where a reboot is required to relinquish ownership, the owner field is cleared, but the current_owner field is not changed. When the owning operating system instance reboots, the console program can clear any current_owner fields for resources that have no owner during initialization.

Detailed Description Text (131):
Until an initial ownership is established (that is, if the owner field is unassigned), CPUs are placed into a HWRPB context decided by the master console, but the HWRPB available bit for the CPU will not be set in any HWRPB. This combination prevents the CPU from joining any operating system instance in SMP operation. When ownership of a CPU is established (the owner field is filled in with a valid partition handle), the CPU will migrate, if necessary, to the owning partition, set the available bit in the HWRPB associated with that partition, and request to join SMP operation of the instance running in that partition, or join the console program in SMP mode. The combination of the present and available bits in the HWRPB tell the operating system instance that the CPU is available for use in SMP operation, and the operating system instance may use these bits to build appropriate per-CPU data structures, and to send a message to the CPU to request it to join SMP operation.

Detailed Description Text (133):
During runtime, an operating system instance can temporarily "loan" a CPU to another partition without changing the nominal ownership of the CPU. The traditional SMP concept of ownership using the HWRPB present and available bits is used to reflect the current execution context of the CPU by modifying the HWRPB and the configuration tree in atomic operations. The current_owner field can further be used by system software in one of the partitions to determine in which partition the CPU is currently executing (other instances can determine the location of a particular CPU by examining the configuration tree.)

Detailed Description Text (137):
The common or shared memory in the system is comprised of memory subsystems 434 and 448 and memory descriptors 438 and 452. These are owned by the community 410. Thus, FIG. 4 describes the layout of the system as it would appear to the operating system instances.

Detailed Description Text (138):
Operating System Characteristics

Detailed Description Text (139):
As previously mentioned, the illustrative computer system can operate with several different operating systems in different partitions. However, conventional operating systems may need to be modified in some aspects in order to make them compatible with the inventive system, depending on how the system is configured. Some sample modifications for the illustrative embodiment are listed below: 1. Instances may need to be modified to include a mechanism for choosing a "primary" CPU in the partition to run the console and be a target for communication from other instances. The selection of a primary CPU can be done in a conventional manner using arbitration mechanisms or other conventional devices. 2. Each instance may need

modifications that allow it to communicate and cooperate with the console program which is responsible for creating a configuration data block that describes the resources available to the partition in which the instance is running. For example, the instance should not probe the underlying hardware to determine what resources are available for usage by the instance. Instead, if it is passed a configuration data block that describes what resources that instance is allowed to access, it will need to work with the specified resources. 3. An instance may need to be capable of starting at an arbitrary physical address and may not be able to reserve any specific physical address in order to avoid conflicting with other operating systems running at that particular address. 4. An instance may need to be capable of supporting multiple arbitrary physical holes in its address space, if it is part of a system configuration in which memory is shared between partitions. In addition, an instance may need to deal with physical holes in its address space in order to support "hot inswap" of memory. 5. An instance may need to pass messages and receive notifications that new resources are available to partitions and instances. More particularly, a protocol is needed to inform an instance to search for a new resource. Otherwise, the instance may never realize that the resource has arrived and is ready for use. 6. An instance may need to be capable of running entirely within its "private memory" if it is used in a system where instances do not share memory. Alternatively, an instance may need to be capable of using physical "shared memory" for communicating or sharing data with other instances running within the computer if the instance is part of a system in which memory is shared. In such a shared memory system, an instance may need to be capable of mapping physical "shared memory" as identified in the configuration tree into its virtual address space, and the virtual address spaces of the "processes" running within that operating system instance. 7. Each instance may need some mechanism to contact another CPU in the computer system in order to communicate with it. 8. An instance may also need to be able to recognize other CPUs that are compatible with its operations, even if the CPUs are not currently assigned to its partition. For example, the instance may need to be able to ascertain CPU parameters, such as console revision number and clock speed, to determine whether it could run with that CPU, if the CPU was re-assigned to the partition in which the instance is running.

Detailed Description Text (141):
Each console program provides a number of callback functions to allow the associated operating system instance to change the configuration of the APMP system, for example, by creating a new community or partition, or altering the ownership of memory fragments. In addition, other callback functions provide the ability to remove a community, or partition, or to start operation on a newly-created partition.

Detailed Description Text (142):
However, callback functions do not cause any changes to take place on the running operating system instances. Any changes made to the configuration tree must be acted upon by each instance affected by the change. The type of action that must take place in an instance when the configuration tree is altered is a function of the type of change, and the operating system instance capabilities. For example, moving an input/output processor from one partition to another may require both partitions to reboot. Changing the memory allocation of fragments, on the other hand, might be handled by an operating system instance without the need for a reboot.

Detailed Description Text (149):
Finally, an operating system instance is booted in at least one of the partitions as indicated in step 510. The first operating system instance to boot creates an APMP database and fills in the entries as described below. APMP databases store information relating to the state of active operating system instances in the system. The routine then finishes in step 512. It should be noted that an instance is not required to participate in an APMP system. The instance can choose not to participate or to participate at a time that occurs well after boot. Those instances which do participate form a "sharing set." The first instance which decides to join a sharing set must create it. There can be multiple sharing sets operating on a single APMP system and each sharing set has its own APMP database.

Detailed Description Text (151):
An operating system instance running on a platform which is also running the APMP

computer system does not necessarily have to be a member of the APMP computer system. The instance can attempt to become a member of the APMP system at any time after booting. This may occur either automatically at boot, or after an operator-command explicitly initiates joining. After the operating system is loaded at boot time, the operating system initialization routine is invoked and examines a stored parameter to see whether it specifies immediate joining and, if so, the system executes a joining routine which is part of the APMP computer system. An operator command would result in an execution of the same routine.

Detailed Description Text (153):
An important data structure supporting the inventive software allocation of resources is the APMP database which keeps track of operating system instances which are members of a sharing set. The first operating system instance attempting to set up the APMP computer system initializes an APMP database, thus creating, or instantiating, the inventive software resource allocations for the initial sharing set. Later instances wishing to become part of the sharing set join by registering in the APMP database associated with that sharing set. The APMP database is a shared data structure containing the centralized information required for the management of shared resources of the sharing set. An APMP database is also initialized when the APMP computer system is re-formed in response to an unrecoverable error.

Detailed Description Text (155):
The initial, header portion of an APMP database is the first part of the APMP database mapped by a joining operating system instance. Portions of the header are accessed before the instance has joined the sharing set, and, in fact, before the instance knows that the APMP computer system exists.

Detailed Description Text (158):
An APMP database is stored in shared memory. The initial fixed portion of N physically contiguous pages occupies the first N pages of one of two memory ranges allocated by the first instance to join during initial partitioning of the hardware. The instance directs the console to store the starting physical addresses of these ranges in the configuration tree. The purpose of allocating two ranges is to permit failover in case of hardware memory failure. Memory management is responsible for mapping the physical memory into virtual address space for the APMP database.

Detailed Description Text (159):
The detailed actions taken by an operating system instance are illustrated in FIG. 6. More specifically, when an operating system instance wishes to become a member of a sharing set, it must be prepared to create the APMP computer system if it is the first instance attempting to "join" a non-existent system. In order for the instance to determine whether an APMP system already exists, the instance must be able to examine the state of shared memory as described above. Further, it must be able to synchronize with other instances which may be attempting to join the APMP system and the sharing set at the same time to prevent conflicting creation attempts. The master console creates the configuration tree as discussed above. Subsequently, a region of memory is initialized by the first, or primary, operating system instance to boot, and this memory region can be used for an APMP database.

Detailed Description Text (161):
The goal of the initial actions taken by all operating system instances is to map the header portion of the APMP database and initialize primitive inter-instance interrupt handling to lay the groundwork for a create or join decision. The routine used is illustrated in FIG. 6 which begins in step 600. The first action taken by each instance (step 602) is to engage memory management to map the initial segment of the APMP database as described above. At this time, the array of node blocks in the second database section is also mapped. Memory management maps the initial and second segments of the APMP database into the primary operating system address space and returns the start address and length. The instance then informs the console to store the location and size of the segments in the configuration tree.

Detailed Description Text (162):
Next, in step 604, the initial virtual address of the APMP database is used to allow the initialization routine to zero interrupt reason masks in the node block assigned to the current instance.

Detailed Description Text (164):
Next, in step 608, the virtual address (VA) of the APMP database is stored in a private cell which is examined by an inter-processor interrupt handler. The handler examines this cell to determine whether to test the per-instance interrupt reason mask in the APMP database header for work to do. If this cell is zero, the APMP database is not mapped and nothing further is done by the handler. As previously discussed, the entire APMP database, including this mask, is initialized so that the handler does nothing before the address is stored. In addition, a clock interrupt handler can examine the same private cell to determine whether to increment the instance-specific heartbeat field for this instance in the appropriate node block. If the private cell is zero, the interrupt handler does not increment the heartbeat field.

Detailed Description Text (170):
Assuming that a new APMP system must be created, the creator instance is responsible for allocating the rest of the APMP database, initializing the header and invoking system services. Assuming the APMP database is locked as described above, the following steps are taken by the creator instance to initialize the APMP system (these steps are shown in FIGS. 7A and 7B): Step 702 the creator instance sets the APMP system state and its node block state to "initializing." Step 704 the creator instance calls a size routine for each system service with the address of its length field in the header. Step 706 the resulting length fields are summed and the creator instance calls memory management to allocate space for the entire APMP database by creating a new mapping and deleting the old mapping. Step 708 the creator instance fills in the offsets to the beginnings of each system service segment. Step 710 the initialization routine for each service is called with the virtual addresses of the APMP database, the service segment and the segment length. Step 712 the creator instance initializes a membership mask to make itself the sole member and increments an incarnation count. It then sets creation time, software version, and other creation parameters. Step 714 the instance then sets itself as its own big and little brother (for heartbeat monitoring purposes as described below). Step 716 the instance then fills in its instance state as "member" and the APMP system state as "operational." Step 718 finally, the instance releases the APMP database lock.

Detailed Description Text (173):
Assuming an instance has the APMP database locked, the following steps are taken by the instance to become a member of an existing APMP system (shown in FIGS. 8A and 8B): Step 802 the instance checks to make sure that its instance name is unique. If another current member has the instance's proposed name, joining is aborted. Step 804 the instance sets the APMP system state and its node block state to "instance joining" Step 806 the instance calls a memory management routine to map the variable portion of the APMP database into its local address space. Step 808 the instance calls system joining routines for each system service with the virtual addresses of the APMP database and its segment and its segment length. Step 810 if all system service joining routines report success, then the instance joining routine continues. If any system service join routine fails, the instance joining process must start over and possibly create a new APMP computer system. Step 812 assuming that success was achieved in step 810, the instance adds itself to the system membership mask. Step 814 the instance selects a big brother to monitor its instance health as set forth below. Step 816 the instance fills in its instance state as "member" and sets a local membership flag. Step 818 the instance releases the configuration database lock.

Detailed Description Text (179):
If a member instance is judged dead, or disinterested, and it has not notified the APMP computer system of its intent to shut down or crash, the instance is removed from the APMP system. This may be done, for example, by setting the "bugcheck" bit in the instance primitive interrupt mask and sending an IP interrupt to all CPU's of the instance. As a rule, shared memory may only be accessed below the hardware priority of the IP interrupt. This insures that if the CPUs in the instance should attempt to execute at a priority below that of the IP interrupt, the IP interrupt will occur first and thus the CPU will see the "bugcheck" bit before any lower priority threads can execute. This insures the operating system instance will crash and not touch shared resources such as memory which may have been reallocated for

other purposes when the instances were judged dead. As an additional or alternative mechanism, a console callback (should one exist) can be invoked to remove the instance. In addition, in accordance with a preferred embodiment, whenever an instance disappears or drops out of the APMP computer system without warning, the remaining instances perform some sanity checks to determine whether they can continue. These checks include verifying that all pages in the APMP database are still accessible, i.e. that there was not a memory failure.

Detailed Description Text (182):
CPU ownership is indicated in a number of ways, in a number of structures dictated by the entity that is managing the resource at the time. In the most basic case, the CPU can be in an unassigned state, available to all partitions that reside in the same sharing set as the CPU. Eventually that CPU is assigned to a specific partition, which may or may not be running an operating system instance. In either case, the partition reflects its ownership to all other partitions through the configuration tree structure, and to all operating system instances that may run in that partition through the AVAILABLE bit in the HWRPB per-CPU flags field.

Detailed Description Text (183):
If the owning partition has no operating system instance running on it, its console is responsible for responding to, and initiating, transition events on the resources within it. The console decides if the resource is in a state that allows it to migrate to another partition or to revert back to the unassigned state.

Detailed Description Text (184):
If, however, there is an instance currently running in the partition, the console relinquishes responsibility for initiating resource transitions and is responsible for notifying the running primary of the instance when a configuration change has taken place. It is still the facilitator of the underlying hardware transition, but control of resource transitions is elevated one level up to the operating system instance. The transfer of responsibility takes place when the primary CPU executes its first instruction outside of console mode in a system boot.

Detailed Description Text (185):
Operating system instances can maintain ownership state information in any number of ways that promote the most efficient usage of the information internally. For example, a hierarchy of state bit vectors can be used which reflect the instance-specific information both internally and globally (to other members sharing an APMP database).

Detailed Description Text (186):
The internal representations are strictly for the use of the instance. They are built up at boot time from the underlying configuration tree and HWRPB information, but are maintained as strict software constructs for the life of the operating system instance. They represent the software view of the partition resources available to the instance, and may--through software rule sets--further restrict the configuration to a subset of that indicated by the physical constructs. Nevertheless, all resources in the partition are owned and managed by the instance--using the console mechanisms to direct state transitions--until that operating system invocation is no longer a viable entity. That state is indicated by halting the primary CPU once again back into console mode with no possibility of returning without a reboot.

Detailed Description Text (189):
During runtime, the current_owner field reflects the partition where a CPU is executing. The AVAILABLE bit in the per-CPU flags field in the HWRPB remains the ultimate indicator of whether a CPU is actually available, or executing, for SMP operation with an operating system instance, and has the same meaning as in conventional SMP systems.

Detailed Description Text (191):
Virtual Resource Management

Detailed Description Text (205):
A software implementation of the above-described embodiment may comprise a series of

computer instructions either fixed on a tangible medium, such as a computer readable media, e.g. a diskette, a CD-ROM, a ROM memory, or a fixed disk, or transmissible to a computer system, via a modem or other interface device over a medium. The medium can be either a tangible medium, including but not limited to optical or analog communications lines, or may be implemented with wireless techniques, including but not limited to microwave, infrared or other transmission techniques. It may also be the Internet. The series of computer instructions embodies all or part of the functionality previously described herein with respect to the invention. Those skilled in the art will appreciate that such computer instructions can be written in a number of programming languages for use with many computer architectures or operating systems. Further, such instructions may be stored using any memory technology, present or future, including, but not limited to, semiconductor, magnetic, optical or other memory devices, or transmitted using any communications technology, present or future, including but not limited to optical, infrared, microwave, or other transmission technologies. It is contemplated that such a computer program product may be distributed as a removable media with accompanying printed or electronic documentation, e.g., shrink wrapped software, pre-loaded with a computer system, e.g., on system ROM or fixed disk, or distributed from a server or electronic bulletin board over a network, e.g., the Internet or World Wide Web.

Detailed Description Text (206):
Although an exemplary embodiment of the invention has been disclosed, it will be apparent to those skilled in the art that various changes and modifications can be made which will achieve some of the advantages of the invention without departing from the spirit and scope of the invention. For example, it will be obvious to those reasonably skilled in the art that, although the description was directed to a particular hardware system and operating system, other hardware and operating system software could be used in the same manner as that described. Other aspects, such as the specific instructions utilized to achieve a particular function, as well as other modifications to the inventive concept are intended to be covered by the appended claims.

Other Reference Publication (2):
Ohmori et al., "System management of MICS-II--a virtual machine complex"; Third USA-Japan Computer Conference Proceedings, San Francisco, Calif., Sep. 1978, pp. 425-429.

Other Reference Publication (4):
Beck, "AAMP: A Multiprocessor Approach for Operating System and Application Migration," Operating Systems Review 24:41-55 (1990).

Other Reference Publication (6):
Rashid et al., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," IEEE Transactions on Computers 37:896-908 (1988).

Other Reference Publication (16):
J. Chapin, et al., Hive: Fault Containment for Shared-Memory Multiprocessors, The 15th ACM Symposium on Operating Systems Principles, 12/95.

Other Reference Publication (18):
Rohit Chandra, et al., Scheduling and Page Migration for Multiprocessor Compute Server, Sixth International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS-V1), 10/94.

Other Reference Publication (31):
C. Hanna, Logical Partitioning Methodologies, CMG (Conference), 1993.

CLAIMS:

1. A computer system having a plurality of system resources including processors, memory and I/O circuitry, the computer system comprising: an interconnection mechanism for electrically interconnecting the processors, memory and I/O circuitry so that each processor has electrical access to all of the memory and at least some of the I/O circuitry; a software mechanism for dividing the system resources into a

plurality of partitions; at least one operating system instance running in each of a plurality of the partitions; and a processor designation storage device that stores designations for each of a plurality of processors to define the association of each processor with one of said plurality of instances; wherein each of the sets for a given instance including a plurality of designations indicating whether a corresponding plurality of said processors has a particular operational status represented by that set, said operational statuses including: a potential status in which the processor are of a type compatible for operation with the given instance; a configure status which indicates whether the processors are under the control of the given instance; and an active status which indicates whether the processors are available for symmetric multiprocessing operation with the given instance.

9. A computer system according to claim 1 wherein at least two of the operating system instances are different operating systems.

13. A computer system having a plurality of system resources including processors, memory and I/O circuitry, the computer system comprising: an interconnection mechanism for electrically interconnecting the processors, memory and I/O circuitry so that each processor has electrical access to all of the memory and at least some of the I/O circuitry; a software mechanism for dividing the system resources into a plurality of partitions; at least one operating system instance running in each of a plurality of the partitions; and means for maintaining, with each instance, a record of processors present in the system, and for recognizing a plurality of different operational statuses of the resources with regard to the instance, said operational status comprising a potential status which designates the resources as being compatible with the instance, a configure status which indicates the resources are under the control of the instance, and an active status which indicates the resources are available to the partition for symmetric multiprocessing operation.

16. A method for constructing a computer system having a plurality of system resources including processors, memory and I/O circuitry, the method comprising the steps of: (a) electrically interconnecting the processors, memory and I/O circuitry so that each processor has electrical access to all of the memory and at least some of the I/O circuitry; (b) dividing the system resources into a plurality of partitions; (c) running at least one operating system instance in a plurality of the partitions; and (d) maintaining, with each instance, a record of processors present in the system, and recognizing a plurality of different operational statuses of the processors with regard to the instance; wherein said operational status comprising a potential status which designates the processors as being compatible with the instance, a configure status which indicates the processors are under the control of the instance, and an active status which indicates the processors are available to the partition for symmetric multiprocessing operation.

19. A method according to claim 16 wherein step (c) comprises the step of: (c1) running at least two different operating system instances in the plurality of partitions, wherein the two different instances are different operating systems.

WEST

End of Result Set

this one

☐ Generate Collection Print

L47: Entry 2 of 2 File: USPT Oct 19, 1999

DOCUMENT-IDENTIFIER: US 5968136 A
TITLE: Apparatus and method for secure device addressing

Detailed Description Text (5):
Operating system 115 includes a set of services 116 that can be invoked by application programs to perform various operating system functions. Herein, the expression "operating system" refers to program code executed to manage hardware resources on behalf of application programs and includes stand-alone virtual machines and operating-system-mounted virtual machines that are used to provide services to application programs.

Detailed Description Text (6):
The set of services 116 is referred to as an "application programming interface" (API) and includes a number of services devoted to device control. Similarly, each device driver 117, 119, 121, 123 and 125 implements a set of device driver services 118, 120, 122, 124 and 126 that can be invoked by operating system 115 to control the attached devices 150, 152, 154, 156 and 158. Each set of device driver services (118, 120, 122, 124 and 126) defines a standard device driver interface, no matter how different the attached hardware devices may be from one another.

Detailed Description Text (8):
As discussed above, operating system 115 does not include device control code for each of the attached devices and instead maintains a data structure identifying device drivers for that purpose. The operating system 115 determines the device driver corresponding to the device control request, indicated by way of example in FIG. 1 to be device driver 119, then invokes device driver 119 by making a call (indicated by arrow 172) to a service provided through device driver interface 120. After being invoked by call 172, device driver 119 may absolutely address registers or data buffers in hardware device 152 directly, or indirectly through another layer of software that provides basic I/O services (BIOS) for fundamental computer components such as display, keyboard and storage media.

Detailed Description Text (9):
In the traditional device driver model of FIG. 1, device driver code (117, 119, 121, 123, 125) is tightly bound to the controlled hardware device (150, 152, 154, 156, 158) and the device driver must be able to address absolute addresses in the system I/O space where the controlled device's registers and data buffers are mapped. Herein, the expression "I/O space" refers to the set of processor addressable locations not already mapped to system memory and includes both memory-mapped locations and I/O port mapped locations. As stated below, system memory may be addressed by either physical or virtual addressing schemes.

Detailed Description Text (10):
It is important to distinguish between addressing I/O space and addressing system memory. Addresses in I/O space are determined by the physical connection of hardware devices to a computer system's address buses. Therefore, unlike program variables or dynamically allocated storage having addresses in system memory, I/O space addresses cannot be remapped (unless, of course, hardware is reconfigured). For example, a program variable used to store data before it is written to a control register could be mapped to a different address in system memory each time the program defining the variable is loaded into system memory by the operating system. By contrast, the

control register itself is mapped to a particular address determined by a physical connection to an address bus and must always be addressed at the particular address. For this reason, I/O space addresses are said to be "absolutely mapped" and a computer program which writes or reads an absolutely mapped address is said to perform "absolute addressing".

Detailed Description Text (20):
FIG. 2 is a flow diagram of a method 200 according to one embodiment of the present invention. At step 205, when the computer is powered on, a boot program stored in firmware is executed to scan the I/O space to determine the identity and location of processor-addressable hardware devices attached to the system. At step 210 the device identity and absolute address information obtained in step 205 is used to generate a database of device information. When this operation is performed in compliance with the standard set forth in IEEE 1275-1994, the database of device information describing each of the addressable devices is stored in a well defined data structure known as a "1275 device tree". In alternate embodiments of the present invention, other databases of device information may be used instead of a 1275 device tree.

Detailed Description Text (27):
FIG. 3 is a block diagram illustrating a software architecture 250 that may be used to implement method 200 of FIG. 2. Architecture 250 is made up of three layers of software: a boot layer 275, memory management layer 280 and access layer 285. Boot layer 275 includes IEEE 1275-1994 firmware 251 and operating system booter 253. Firmware 251 is executed to scan the I/O space of the computer system in which it is installed to determine the identity and absolute address of addressable devices in the computer system. Operating system booter 253, which also may be provided in a non-volatile storage device, includes code that is executed to load an operating system into the computer system's memory.

Detailed Description Text (28):
Memory management layer 280 includes a device tree database 255 (designated "1275 device tree" in FIG. 2), a platform manager 257 and a memory management package 259. The identity and absolute address information obtained during execution of firmware 251 is copied to device tree database 255 for later use by platform manager 257. This is indicated by arrow 271. As discussed above, platform manager 257 includes an API that provides procedures that may be called to obtain and activate, respectively, a memory access object. Thus, if during execution of code in memory management package 259, it is necessary to access an absolute address, a procedure call (indicated by arrow 272) is issued to platform manager 257 requesting a memory access object. Based on a passed parameter received in procedure call 272, platform manager 257 examines the 1275 device tree 255 (as indicated by arrow 273) to locate the absolute address. Platform manager 257 then encapsulates the absolute address in a memory object and returns a reference to the object to the calling code within memory management package 259. As stated above, the access methods of the memory access object must be activated before they can be invoked to perform absolute addressing. Thus, code in memory management package 259 is executed to issue procedure call 277 to platform manager 257. After confirming that the calling code is included within trusted memory package 259, the platform manager procedure invoked by call 277 activates the access methods of the memory access object.

Detailed Description Text (31):
FIG. 4 depicts a device tree 351 used to represent a computer architecture 301. As stated above, device tree 351 is a data structure constructed by boot firmware to provide information about the hardware attached to a computer system. "Open Firmware" is the name given to non-proprietary boot firmware that can construct a device tree according to the IEEE 1275-1994 standard for a number of different hardware platforms. For a given hardware platform, each system bus corresponds to an interior node of the device tree and the devices coupled to the system bus are represented by child nodes of the interior node. This way, the structure of the device tree reflects the structure of the underlying hardware. For example, bus 305 of computer system 301 corresponds to interior node 355 of device tree 351, and attached devices 307, 309, 311, 313, 315 and 317 (i.e., microprocessor, memory, serial I/O, parallel I/O, storage media and display, respectively) correspond to child nodes 357, 359, 361, 363, 365 and 367 of interior node 305. It will be

appreciated that numerous other devices could be attached to bus 305 and that there may be multiple system buses in a more complex architecture.

Detailed Description Text (32):
According to the IEEE 1275-1994 standard, devices plugged into expansion slots on a computer bus report their characteristics to Open Firmware via a device interface. Open firmware then stores the reported information in a device tree node established for the reporting device. The reported information will typically include the device name, model, revision level, device type, register locations, interrupt levels, supported features and any other information significant to the operation of the reporting device. Device tree nodes are also established for permanently installed devices and the set of information used to describe a particular device can be extended to support new types of devices having new characteristics.

Detailed Description Text (33):
As FIG. 5 illustrates, absolute addresses stored in the device tree 351 are used in the construction of memory access objects 405, 415, 425 and 435. Open Firmware includes a client interface defined by the IEEE 1275-1994 standard to allow operating systems and other programs to access the device tree. In a preferred embodiment of the present invention, the operating system invokes services provided by the Open Firmware client interface to identify absolute addresses corresponding to attached devices. For each absolute address identified, a memory access object may be instantiated and the absolute address encapsulated in the memory access object.

Detailed Description Text (35):
The size and format of an absolute address is usually dependent on a processor or bus included in the hardware platform. Even within a single hardware platform, some absolute addresses may be memory mapped while others may be I/O port mapped. Thus, when an operating system including memory access objects according to the present invention is executing on a processor having a 32-bit address width, a memory access object capable of wrapping a range of 32-bit addresses may be used. When executing on a processor having a 64-bit address width, a memory access object capable of wrapping a range of 64-bit addresses is necessary. In a preferred embodiment, an address class tool-kit is provided to allow different sized and formatted absolute addresses to be wrapped in objects. Once a class has been defined it can be referenced by the platform manager of FIG. 3 to build a memory access object.

Detailed Description Text (37):
FIG. 6 illustrates a device driver model according to the present invention. Application program 105, operating system 515 and device drivers 532, 534, 536 and 538 are all loaded into system memory. After application program 105 invokes an operating system service provided in the operating system's API 116 to perform device control (as indicated by arrow 171), the operating system identifies the device driver responsible for performing the requested device control operation (in this case device driver 532), then calls a service provided by the device driver interface 118 of device driver 532 as indicated by arrow 172. So far, events have been as described above in reference to the traditional device driver model of FIG. 1. However, instead of accessing the attached hardware device at an absolute address maintained in the device driver 532 code or pointer variables, device driver 532 invokes a method made public in memory access object 505. Depending on the operation originally requested by application program 105, the method of memory access object 505 may write or read the absolute address of a register or data buffer in the controlled device, in this case storage media 150. Registers and buffers within the other attached devices (Display 152, Sound card 154, Print driver 156, Modem 158) can be read and written in a similar fashion by invoking methods in other operating system memory access objects (506, 507, 508).

Detailed Description Text (43):
Memory 309 may include both system memory (e.g., random access memory) and non-volatile storage such as a semiconductor read-only-memory, hard disk-drive, floppy disk-drive, optical disk-drive or any other computer-readable medium. When power is applied to the computer system 600, program code defining an operating system is loaded from non-volatile storage into system memory by processor 307 or another device, such as a direct memory access controller (not shown), having access

to memory 309. Sequences of instructions comprised by the operating system are then executed by processor 307 to load other computer programs and portions of computer programs into system memory from non-volatile storage. The present invention may be embodied in a sequence of instructions which can be stored in a computer-readable medium and executed by processor 307. It will be appreciated that both system memory and non-volatile storage may be used to effectuate a virtual memory. In that case, sequences of instructions defining a portion of the operating system or an application program may be kept in non-volatile storage and then moved to system memory when required for execution.

Other Reference Publication (1):
"The Smalltalk-80 Virtual Machine", Glenn Krasner, BYTE Publications, Inc., Aug. 1991, pp. 300-320.

Other Reference Publication (4):
"Welcome to the Open Firmware Home Page", URL http//www.firmworks.com, Dec. 23, 1996, 7 pp.

Other Reference Publication (5):
"What is Open Firmware?", URL http://www.firmworks.com, Oct. 3, 1996, 6 pp.

WEST

3

End of Result Set

☐ Generate Collection Print

L12: Entry 3 of 3 File: USPT Mar 25, 2003

DOCUMENT-IDENTIFIER: US 6538669 B1
TITLE: Graphical user interface for configuration of a storage system

Detailed Description Text (41):
The ISAN server 102A has four separate 64-bit 66 MHz PCI busses 200A-D. Many different configurations of storage devices and network interfaces in the slots of the PCI busses are possible. In one embodiment, the PCI busses are divided into two groups: the SSD PCI busses 200A-B and the interface PCI busses 200C-D. Each group has two busses that are designated by the terms upper and lower. The upper and lower busses in each group can be configured to provide redundant services. For example, the lower SSD PCI bus 200B has the same configuration as the upper SSD PCI bus 200A.

Detailed Description Text (80):
FIG. 10 illustrates a partition ISM 750. The partition ISM 750 includes an interface 751 which receives internal communications from other driver modules, and an interface 752 which also communicates with other driver modules. The ISM 750 includes logic processes 753, data structures for storing a base address 754 and a limit address 755, and a drive interface 756. The partition logic process 753 configures the subject storage device identified by the drive process 756, using a logical partitioning function useful for a variety of storage management techniques, so that the physical device appears as more than one logical device in the virtual circuits. In this example, the partition ISM 750 is an instance of a class "partition," and given device identifier 10400.

Detailed Description Text (107):
The connection option such as the network interface 146 over which the storage transaction is received is coupled to a hardware device driver. The hardware device driver receives the storage transaction and depending on the protocol, dispatches it to an appropriate virtual device for handling that protocol.

WEST

(4)

Generate Collection Print

L11: Entry 4 of 8 File: USPT May 21, 2002

DOCUMENT-IDENTIFIER: US 6393495 B1
TITLE: Modular virtualizing device driver architecture

Brief Summary Text (3):
The present invention is generally related to the design of device drivers utilized in computer operating systems to define and establish an interface between the core operating system and typically hardware devices and, in particular, to a modular device driver architecture providing a virtualized, context switchable interface environment within which to operate typically hardware devices in support of operating system functions, specifically including information display functions.

Detailed Description Text (69):
The operating system interface objects, including a GDI object 120, Direct Draw object 122, Direct 3D object 124 and Shell object 126, represent a set of objects that are logically partitioned from one another by the definition of the partial APIs that they present to the operating system layer 54. The revision of existing API call support and the addition of new API calls to any particular operating system interface object has, by design, essentially no impact on the implementation or operation of other operating system interface objects. Further, support for a new partial API, either as newly defined by the operating system layer 54 or to support calls originated directly from an application 60, can be readily provided through definition of a corresponding operating system interface object and encapsulated operating system interface module. However, if a new or revised API call involves or requires a significantly different function than any of the other API calls supported by the existing operating system interface objects, additional library routines may need to be added to the shell library 72'.

Detailed Description Text (70):
The shell library 72' is logically partitioned from operating system interface modules to provide a library of routines that serve to establish a set of common, or virtualized, support functions usable by the full set of operating system interface objects. Preferably, each of the operating system interface objects is functionally constrained to (1) support a well-defined API call set, (2) provide for parameter validation for each supported API call, (3) potentially manage a private data space for data objects that are desired to persist across context changes in the operation of the device driver 50, (4) issue a sequence of one or more virtualized calls to the shell library 72' to functionally implement each of the API calls supported by the object and, finally, (5) format and return data to the operating system layer 54 in a manner appropriate for each API call supported by the object.

L9: Entry 4 of 13 File: USPT Apr 1, 2003

DOCUMENT-IDENTIFIER: US 6542926 B2
TITLE: Software partitioned multi-processor system with flexible resource sharing levels

Abstract Text (1):
Multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources is subdivided by software into multiple partitions, each running a distinct copy, or instance, of an operating system. Each of the partitions has access to its own physical resources plus resources designated as shared. The partitioning is performed by assigning all resources with a configuration tree. None, some, or all, resources may be designated as shared among multiple partitions. Each individual operating instance will generally be assigned the resources it needs to execute independently and these resources will be designated as "private." Other resources, particularly memory, can be assigned to more than one instance and shared. Shared memory is cache coherent so that instances may be tightly coupled, and may share resources that are normally allocated to a single instance. This allows previously distributed user or operating system applications which usually must pass messages via an external interconnect to operate cooperatively in the shared memory without the need for either an external interconnect or message passing. Examples of application that could take advantage of this capability include distributed lock managers and cluster interconnects. Newly-added resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration.

Brief Summary Text (2):
This invention relates to multiprocessor computer architectures in which processors and other computer hardware resources are grouped in partitions, each of which has an operating system instance and, more specifically, to methods and apparatus for sharing resources in a variety of configurations between partitions.

Brief Summary Text (5):
Traditionally, computing speed has been addressed by using a "shared nothing" computing architecture where data, business logic, and graphic user interfaces are distinct tiers and have specific computing resources dedicated to each tier. Initially, a single central processing unit was used and the power and speed of such a computing system was increased by increasing the clock rate of the single central processing unit. More recently, computing systems have been developed which use several processors working as a team instead one massive processor working alone. In this manner, a complex application can be distributed among many processors instead of waiting to be executed by a single processor. Such systems typically consist of several central processing units (CPUs) which are controlled by a single operating system. In a variant of a multiple processor system called "symmetric multiprocessing" or SMP, the applications are distributed equally across all processors. The processors also share memory. In another variant called "asymmetric multiprocessing" or AMP, one processor acts as a "master" and all of the other processors act as "slaves." Therefore, all operations, including the operating system, must pass through the master before being passed onto the slave processors. These multiprocessing architectures have the advantage that performance can be increased by adding additional processors, but suffer from the disadvantage that the software running on such systems must be carefully written to take advantage of the

multiple processors and it is difficult to scale the software as the number of processors increases. Current commercial workloads do not scale well beyond 8-24 CPUs as a single SMP system, the exact number depending upon platform, operating system and application mix.

Brief Summary Text (8):
The availability and maintainability issues were addressed by a "shared everything" model in which a large centralized robust server that contains most of the resources is networked with and services many small, uncomplicated client network computers. Alternatively, "clusters" are used in which each system or "node" has its own memory and is controlled by its own operating system. The systems interact by sharing disks and passing messages among themselves via some type of communication network. A cluster system has the advantage that additional systems can easily be added to a cluster. However, networks and clusters suffer from a lack of shared memory and from limited interconnect bandwidth which places limitations on performance.

Brief Summary Text (9):
In many enterprise computing environments, it is clear that the two separate computing models must be simultaneously accommodated and each model optimized. Further, it is highly desirable to be able to modify computer configurations "on the fly" without rebooting any of the systems. Several prior art approaches have been used to attempt this accommodation. For example, a design called a "virtual machine" or VM developed and marketed by International Business Machines Corporation, Armonk, N.Y., uses a single physical machine, with one or more physical processors, in combination with software which simulates multiple virtual machines. Each of those virtual machines has, in principle, access to all the physical resources of the underlying real computer. The assignment of resources to each virtual machine is controlled by a program called a "hypervisor". There is only one hypervisor in the system and it is responsible for all the physical resources. Consequently, the hypervisor, not the other operating systems, deals with the allocation of physical hardware. The hypervisor intercepts requests for resources from the other operating systems and deals with the requests in a globally-correct way.

Brief Summary Text (11):
In addition, the CPUs logically assigned to each partition can be turned "on" and "off" dynamically via normal operating system operator commands without re-boot. The only limitation is that the number of CPUs active at system initialization is the maximum number of CPUs that can be turned "on" in any partition.

Brief Summary Text (15):
The Hive Project conducted at Stanford University uses an architecture which is structured as a set of cells. When the system boots, each cell is assigned a range of nodes, each having memory and I/O devices, that the cell owns throughout execution. Each cell manages the processors, memory and I/O devices on those nodes as if it were an independent operating system. The cells cooperate to present the illusion of a single system to user-level processes.

Brief Summary Text (17):
A system called "Cellular IRIX" developed and marketed by Silicon Graphics Inc. Mountain View, Calif., supports modular computing by extending traditional symmetric multiprocessing systems. The Cellular IRIX architecture distributes global kernel text and data into optimized SMP-sized chunks or "cells". Cells represent a control domain consisting of one or more machine modules, where each module consists of processors, memory, and I/O. Applications running on these cells rely extensively on a full set of local operating system services, including local copies of operating system text and kernel data structures, bit only one instance of the operating system exists on the entire system. Inter-cell coordination allows application images to directly and transparently utilize processing, memory and I/O resources from other cells without incurring the overhead of data copies or extra context switches.

Brief Summary Text (18):
Another existing architecture called NUMA-Q developed and marketed by Sequent Computer Systems, Inc., Beaverton, Oregon uses "quads", or a group of four processors per portion of memory, as the basic building block for NUMA-Q SMP nodes.

Adding I/O to each quad further improves performance. Therefore, the NUMA-Q architecture not only distributes physical memory but puts a predetermined number of processors and PCI slots next to each processor. The memory in each quad is not local memory in the traditional sense. Rather, it is a portion of the physical memory address space and has a specific address range. The address map is divided evenly over memory, with each quad containing a contiguous portion of address space. Only one copy of the operating system is running and, as in any SMP system, it resides in memory and runs processes without distinction and simultaneously on one or more processors.

Brief Summary Text (21):
In accordance with the principles of the present invention, multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources is subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system. Each of the partitions has access to its own physical resources plus resources designated as shared. In accordance with one embodiment, the partitioning is performed by assigning resources using a configuration data structure such as a configuration tree.

Brief Summary Text (23):
Newly-added resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration.

Detailed Description Text (2):
A computer platform constructed in accordance with the principles of the present invention is a multi-processor system capable of being partitioned to allow the concurrent execution of multiple instances of operating system software. The system does not require hardware support for the partitioning of its memory, CPUs and I/O subsystems, but some hardware may be used to provide additional hardware assistance for isolating faults, and minimizing the cost of software engineering. The following specification describes the interfaces and data structures required to support the inventive software architecture. The interfaces and data structures described are not meant to imply a specific operating system must be used, or that only a single type of operating system will execute concurrently. Any operating system which implements the software requirements discussed below can participate in the inventive system operation.

Detailed Description Text (6):
An APMP computer system 200 constructed in accordance with the principles of the present invention from a software view is illustrated in FIG. 2. In this system, the hardware components have been allocated to allow concurrent execution of multiple operating system instances 208, 210, 212.

Detailed Description Text (7):
In a preferred embodiment, this allocation is performed by a software program called a "console" program, which, as will hereinafter be described in detail, is loaded into memory at power up. Console programs are shown schematically in FIG. 2 as programs 213, 215 and 217. The console program may be a modification of an existing administrative program or a separate program which interacts with an operating system to control the operation of the preferred embodiment. The console program does not virtualize the system resources, that is, it does not create any software layers between the running operating systems 208, 210 and 212 and the physical hardware, such as memory and I/O units (not shown in FIG. 2.) Nor is the state of the running operating systems 208, 210 and 212 swapped to provide access to the same hardware. Instead, the inventive system logically divides the hardware into partitions. It is the responsibility of operating system instance 208, 210, and 212 to use the resources appropriately and provide coordination of resource allocation and sharing. The hardware platform may optionally provide hardware assistance for the division of resources, and may provide fault barriers to minimize the ability of an operating system to corrupt memory, or affect devices controlled by another operating system copy.

Detailed Description Text (8):
The execution environment for a single copy of an operating system, such as copy 208 is called a "partition" 202, and the executing operating system 208 in partition 202 is called "instance" 208. Each operating system instance is capable of booting and running independently of all other operating system instances in the computer system, and can cooperatively take part in sharing resources between operating system instances as described below.

Detailed Description Text (9):
In order to run an operating system instance, a partition must include a hardware restart parameter block (HWRPB), a copy of a console program, some amount of memory, one or more CPUs, and at least one I/O bus which must have a dedicated physical port for the console. The HWRPB is a configuration block which is passed between the console program and the operating system.

Detailed Description Text (10):
Each of console programs 213, 215 and 217, is connected to a console port, shown as ports 214, 216 and 218, respectively. Console ports, such as ports 214, 216 and 218, generally come in the form of a serial line port, or attached graphics, keyboard and mouse options. For the purposes of the inventive computer system, the capability of supporting a dedicated graphics port and associated input devices is not required, although a specific operating system may require it. The base assumption is that a serial port is sufficient for each partition. While a separate terminal, or independent graphics console, could be used to display information generated by each-console, preferably the serial lines 220, 222 and 224, can all be connected to a single multiplexer 226 attached to a workstation, PC, or LAT 228 for display of console information.

Detailed Description Text (12):
Partitions can be "initialized" or "uninitialized." An initialized partition has sufficient resources to execute an operating system instance, has a console program image loaded, and a primary CPU available and executing. An initialized partition may be under control of a console program, or may be executing an operating system instance. In an initialized state, a partition has full ownership and control of hardware components assigned to it and only the partition itself may release its components.

Detailed Description Text (14):
An uninitialized partition is a partition which has no primary CPU executing either under control of a console program or an operating system. For example, a partition may be uninitialized due to a lack of sufficient resources at power up to run a primary CPU, or when a system administrator is reconfiguring the computer system. When in an uninitialized state, a partition may reassign its hardware components and may be deleted by another partition. Unassigned resources may be assigned by any partition.

Detailed Description Text (15):
Partitions may be organized into "communities" which provide the basis for grouping separate execution contexts to allow cooperative resource sharing. Partitions in the same community can share resources. Partitions that are not within the same community cannot share resources. Resources may only be manually moved between partitions that are not in the same community by the system administrator by de-assigning the resource (and stopping usage), and manually reconfiguring the resource. Communities can be used to create independent operating system domains, or to implement user policy for hardware usage. In FIG. 2, partitions 202 and 204 have been organized into community 230. Partition 206 may be in its own community 205. Communities can be constructed using the configuration tree described below and may be enforced by hardware.

Detailed Description Text (18):
More specifically, when the APMP system is powered up, a CPU will be selected as a primary CPU in a conventional manner by hardware which is specific to the platform on which the system is running. The primary CPU then loads a copy of a console program into memory. This console copy is called a "master console" program. The primary CPU initially operates under control of the master console program to

perform testing and checking assuming that there is a single system which owns the entire machine. Subsequently, a set of environment variables are loaded which define the system partitions. Finally, the master console creates and initializes the partitions based on the environment variables. In this latter process the master console operates to create the configuration tree, to create additional HWRPB data blocks, to load the additional console program copies, and to start the CPUs on the alternate HWRPBs. Each partition then has an operating system instance running on it, which instance cooperates with a console program copy also running in that partition. In an unconfigured APMP system, the master console program will initially create a single partition containing the primary CPU, a minimum amount of memory, and a physical system administrator's console selected in a platform-specific way. Console program commands will then allow the system administrator to create additional partitions, and configure I/O buses, memory, and CPUs for each partition.

Detailed Description Text (19):
After associations of resources to partitions have been made by the console program, the associations are stored in non-volatile RAM to allow for an automatic configuration of the system during subsequent boots. During subsequent boots, the master console program must validate the current configuration with the stored configuration to handle the removal and addition of new components. Newly-added components are placed into an unassigned state, until they are assigned by the system administrator. If the removal of a hardware component results in a partition with insufficient resources to run an operating system, resources will continue to be assigned to the partition, but it will be incapable of running an operating system instance until additional new resources are allocated to it.

Detailed Description Text (20):
As previously mentioned, the console program communicates with an operating system instance by means of an HWRPB which is passed to the operating system during operating system boot up. The fundamental requirements for a console program are that it should be able to create multiple copies of HWRPBs and itself. Each HWRPB copy created by the console program will be capable of booting an independent operating system instance into a private section of memory and each operating system instance booted in this manner can be identified by a unique value placed into the HWRPB. The value indicates the partition, and is also used as the operating system instance ID.

Detailed Description Text (21):
In addition, the console program is configured to provide a mechanism to remove a CPU from the available CPUs within a partition in response to a request by an operating system running in that partition. Each operating system instance must be able to shutdown, halt, or otherwise crash in a manner that control is passed to the console program. Conversely, each operating system instance must be able to reboot into an operational mode, independently of any other operating system instance.

Detailed Description Text (22):
Each HWRPB which is created by a console program will contain a CPU slot-specific database for each CPU that is in the system, or that can be added to the system without powering the entire system down. Each CPU that is physically present will be marked "present", but only CPUs that will initially execute in a specific partition will be marked "available" in the HWRPB for the partition. The operating system instance running on a partition will be capable of recognizing that a CPU may be available at some future time by a present (PP) bit in a per-CPU state flag fields of the HWRPB, and can build data structures to reflect this. When set, the available (PA) bit in the per-CPU state flag fields indicates that the associated CPU is currently associated with the partition, and can be invited to join SMP operation.

Detailed Description Text (24):
As previously mentioned, the master console program creates a configuration tree which represents the hardware configuration, and the assignment of each component in the system to each partition. Each console program then identifies the configuration tree to its associated operating system instance by placing a pointer to the tree in the HWRPB.

Detailed Description Text (26):
The master console may generate a single copy of the tree which copy is shared by all operating system instances, or it may replicate the tree for each instance. A single copy of the tree has the disadvantage that it can create a single point of failure in systems with independent memories. However, platforms that generate multiple tree copies require the console programs to be capable of keeping changes to the tree synchronized.

Detailed Description Text (27):
The configuration tree comprises multiple nodes including root nodes, child nodes and sibling nodes. Each node is formed of a fixed header and a variable length extension for overlaid data structures. The tree starts with a tree root node 302 representing the entire system box, followed by branches that describe the hardware configuration (hardware root node 304), the software configuration (software root node 306), and the minimum partition requirements (template root node 308.) In FIG. 3, the arrows represent child and sibling relationships. The children of a node represent component parts of the hardware or software configuration. Siblings represent peers of a component that may not be related except by having the same parent. Nodes in the tree 300 contain information on the software communities and operating system instances, hardware configuration, configuration constraints, performance boundaries and hot-swap capabilities. The nodes also provide the relationship of hardware to software ownership, or the sharing of a hardware component.

Detailed Description Text (28):
The nodes are stored contiguously in memory and the address offset from the tree root node 302 of the tree 300 to a specific node forms a "handle" which may be used from any operating system instance to unambiguously identify the same component on any operating system instance. In addition, each component in the inventive computer system has a separate ID. This may illustratively be a 64-bit unsigned value. The ID must specify a unique component when combined with the type and subtype values of the component. That is, for a given type of component, the ID must identify a specific component. The ID may be a simple number, for example the CPU ID, it may be some other unique encoding, or a physical address. The component ID and handle allow any member of the computer system to identify a specific piece of hardware or software. That is, any partition using either method of specification must be able to use the same specification, and obtain the same result.

Detailed Description Text (29):
As described above, the inventive computer system is composed of one or more communities which, in turn, are composed of one or more partitions. By dividing the partitions across the independent communities, the inventive computer system can be placed into a configuration in which sharing of devices and memory can be limited. Communities and partitions will have IDs which are densely packed. The hardware platform will determine the maximum number of partitions based on the hardware that is present in the system, as well as having a platform maximum limit. Partition and community IDs will never exceed this value during runtime. IDs will be reused for deleted partitions and communities. The maximum number of communities is the same as the maximum number of partitions. In addition, each operating system instance is identified by a unique instance identifier, for example a combination of the partition ID plus an incarnation number.

Detailed Description Text (32):
Hardware components place configuration constraints on how ownership may be divided. A "config" handle in the configuration tree node associated with each component determines if the component is free to be associated anywhere in the computer system by pointing to the hardware root node 304. However, some hardware components may be bound to an ancestor node and must be configured as part of this node. Examples of this are CPUs, which may have no constraints on where they execute, but which are a component part of a system building block (SBB), such as SBBs 322 or 324. In this case, even though the CPU is a child of the SBB, its config handle will point to the hardware root node 304. An I/O bus, however, may not be able to be owned by a partition other than the partition that owns its I/O processor. In this case, the configuration tree node representing the I/O bus would have a config handle pointing to the I/O processor. Because the rules governing hardware configuration are

platform specific, this information is provided to the operating system instances by the config handle.

Detailed Description Text (36):
The configuration tree 300 may extend to the level of device controllers, which will allow the operating system to build bus and device configuration tables without probing the buses. However, the tree may also end at any level, if all components below it cannot be configured independently. System software will still be required to probe for bus and device information not provided by the tree.

Detailed Description Text (38):
The minimal component requirements for a partition are provided by the information contained in the template root node 308. The template root node 308 contains nodes, 316, 318 and 320, representing the hardware components that must be provided to create a partition capable of execution of a console program and an operating system instance. Configuration editors can use this information as the basis to determine what types, and how many resources must be available to form a new partition.

Detailed Description Text (39):
During the construction of a new partition, the template subtree will be "walked", and, for each node in the template subtree, there must be a node with the same type and subtype owned by the new partition so that it will be capable of loading a console program and booting an operating system instance. If there are more than one node of the same type and subtype in the template tree, there must also be multiple nodes in the new partition. The console program will use the template to validate that a new partition has the minimum requirements prior to attempting to load a console program and initialize operation.

Detailed Description Text (41):
The total size in bytes of the memory allocated for the configuration tree is located in the first quadword of the header. The size is guaranteed to be in multiples of the hardware page size. The second quadword of the header is reserved for a checksum. In order to examine the configuration tree, an operating system instance maps the tree into its local address space. Because an operating system instance may map this memory with read access allowed for all applications, some provision must be made to prevent a non-privileged application from gaining access to console data to which it should not have access. Access may be restricted by appropriately allocating memory. For example, the memory may be page aligned and allocated in whole pages. Normally, an operating system instance will map the first page of the configuration tree, obtain the tree size, and then remap the memory allocated for configuration tree usage. The total size may include additional memory used by the console for dynamic changes to the tree.

Detailed Description Text (47):
This field is used as a simple lock by software wishing to inhibit changes to the structure of the tree, and the software configuration. When this value is -1 (all bits on) the tree is unlocked; when the value is >=0 the tree is locked. This field is modified using atomic operations. The caller of the lock routine passes a partition ID which is written to the lock field. This can be used to assist in fault tracing, and recovery during crashes. transient_level This field is incremented at the start of a tree update. current_level This field is updated at the completion of a tree update. console_req This field specifies the memory required in bytes for the console in the base memory segment of a partition. min_alloc This field holds the minimum size of a memory fragment, and the allocation unit (fragments size must be a multiple of the allocation). It must be a power of 2. min_align This field holds the alignment requirements for a memory fragment. It must be a power of 2. base_alloc This field specifies the minimum memory in bytes (including console_req) needed for the base memory segment for a partition. This is where the console, console structures, and operating system will be loaded for a partition. It must be greater or equal to minAlloc and a multiple of minAlloc. base_align This field holds the alignment requirement for the base memory segment of a partition. It must be a power of 2, and have an alignment of at least min_alig005 n. max_phys_address The field holds the calculated largest physical address that could exist on the system, including memory subsystems that are not currently powered on and available. mem_size This field holds the total memory currently in system. platform_type This

field stores the type of platform taken from a field in the HWRPB. platform_name This field holds an integer offset from the base of the tree root node to a string representing the name of the platform. primary_instance This field stores the partition ID of the first operating system instance. first_free This field holds the offset from the tree root node to the first free byte of memory pool used for new nodes. high_limit This field holds the highest address at which a valid node can be located within the configuration tree. It is used by callbacks to validate that a handle is legal. lookaside This field is the handle of a linked list of nodes that have been deleted, and that may be reclaimed. When a community or partition are deleted, the node is linked into this list, and creation of a new partition or community will look at this list before allocating from free pool. available This field holds the number of bytes remaining in the free pool pointed to by the first_free field. max_partitions This field holds the maximum number of partitions computed by the platform based on the amount of hardware resources currently available. partitions This field holds an offset from the base of the root node to an array of handles.

Detailed Description Text (49):
A community provides the basis for the sharing of resources between partitions. While a hardware component may be assigned to any partition in a community, the actual sharing of a device, such as memory, occurs only within a community. The community node 310 contains a pointer to a control section, called an APMP database, which allows the operating system instances to control access and membership in the community for the purpose of sharing memory and communications between instances. The APMP database and the creation of communities are discussed in detail below. The configuration ID for the community is a signed 16-bit integer value assigned by the console program. The ID value will never be greater than the maximum number of partitions that can be created on the platform.

Detailed Description Text (50):
A partition node, such as node 312 or 314, represents a collection of hardware that is capable of running an independent copy of the console program, and an independent copy of an operating system. The configuration ID for this node is a signed 16-bit integer value assigned by the console. The ID will never be greater than the maximum number of partitions that can be created on the platform. The node has the definition:

Detailed Description Text (51):
The defined fields have the definitions: hwrpb This field holds the physical address of the hardware restart parameter block for this partition. To minimize changes to the HWRPB, the HWRPB does not contain a pointer to the partition, or the partition ID. Instead, the partition nodes contain a pointer to the HWRPB. System software can then determine the partition ID of the partition in which it is running by searching the partition nodes for the partition which contains the physical address of its HWRPB. incarnation This field holds a value which is incremented each time the primary CPU of the partition executes a boot or restart operation on the partition. priority This field holds a partition priority. os_type This field holds a value which indicates the type of operating system that will be loaded in the partition. partition_reserved.sub.-- 1 This field is reserved for future use. instance_name_format This field holds a value that describes the format of the instance name string. instance_name This field holds a formatted string which is interpreted using the instance_name_format field. The value in this field provides a high-level path name to the operating system instance executing in the partition. This field is loaded by system software and is not saved across power cycles. The field is cleared at power up and at partition creation and deletion.

Detailed Description Text (55):
A memory controller node (such as nodes 336 or 350) is used to express a physical hardware component, and its owner is typically the partition which will handle errors, and initialization. Memory controllers cannot be assigned to communities, as they require a specific operating system instance for initialization, testing and errors. However, a memory description, defined by a memory descriptor node, may be split into "fragments" to allow different partitions or communities to own specific memory ranges within the memory descriptor. Memory is unlike other hardware resources in that it may be shared concurrently, or broken into "private" areas.

Each memory descriptor node contains a list of subset ranges that allow the memory to be divided among partitions, as well as shared between partitions (owned by a community). A memory descriptor node (such as nodes 338 or 352) is defined as:

Detailed Description Text (60):
Fragments can have minimum allocation sizes and alignments provided in the tree root node 302. The base memory for a partition (the fragments where the console and operating system will be loaded) may have a greater allocation and alignment than other fragments (see the tree root node definition above). If the owner field of the memory descriptor node is a partition, then the fragments can only be owned by that partition.

Detailed Description Text (61):
FIG. 4 illustrates the configuration tree shown in FIG. 3 when it is viewed from a perspective of ownership. The console program for a partition relinquishes ownership and control of the partition resources to the operating system instance running in that partition when the primary CPU for that partition starts execution. The concept of "ownership" determines how the hardware resources and CPUs are assigned to software partitions and communities. The configuration tree has ownership pointers illustrated in FIG. 4 which determine the mapping of hardware devices to software such as partitions (exclusive access) and communities (shared access). An operating system instance uses the information in the configuration tree to determine to which hardware resources it has access and reconfiguration control.

Detailed Description Text (62):
Passive hardware resources which have no owner are unavailable for use until ownership is established. Once ownership is established by altering the configuration tree, the operating system instances may begin using the resources. When an instance makes an initial request, ownership can be changed by causing the owning operating system to stop using a resource or by a console program taking action to stop using a resource in a partition where no operating system instance is executing. The configuration tree is then altered to transfer ownership of the resource to another operating system instance. The action required to cause an operating system to stop using a hardware resource is operating system specific, and may require a reboot of the operating system instances affected by the change.

Detailed Description Text (65):
When a resource is de-assigned, the owner may decide to de-assign the owner field, or both the owner and current_owner fields. The decision is based on the ability of the owning operating system instance running in the partition to discontinue the use of the resource prior to de-assigning ownership. In the case where a reboot is required to relinquish ownership, the owner field is cleared, but the current_owner field is not changed. When the owning operating system instance reboots, the console program can clear any current_owner fields for resources that have no owner during initialization.

Detailed Description Text (68):
Until an initial ownership is established (that is, if the owner field is unassigned), CPUs are placed into a HWRPB context decided by the master console, but the HWRPB available bit for the CPU will not be set in any HWRPB. This combination prevents the CPU from joining any operating system instance in SMP operation. When ownership of a CPU is established (the owner field is filled in with a valid partition handle), the CPU will migrate, if necessary, to the owning partition, set the available bit in the HWRPB associated with that partition, and request to join SMP operation of the instance running in that partition, or join the console program in SMP mode. The combination of the present and available bits in the HWRPB tell the operating system instance that the CPU is available for use in SMP operation, and the operating system instance may use these bits to build appropriate per-CPU data structures, and to send a message to the CPU to request it to join SMP operation.

Detailed Description Text (70):
During runtime, an operating system instance can temporarily "loan" a CPU to another partition without changing the nominal ownership of the CPU. The traditional SMP concept of ownership using the HWRPB present and available bits is used to reflect the current execution context of the CPU by modifying the HWRPB and the

configuration tree in atomic operations. The current_owner field can further be used by system software in one of the partitions to determine in which partition the CPU is currently executing (other instances can determine the location of a particular CPU by examining the configuration tree.)

Detailed Description Text (74):
The common or shared memory in the system is comprised of memory subsystems 434 and 448 and memory descriptors 438 and 452. These are owned by the community 410. Thus, FIG. 4 describes the layout of the system as it would appear to the operating system instances.

Detailed Description Text (75):
Operating System Characteristics

Detailed Description Text (76):
As previously mentioned, the illustrative computer system can operate with several different operating systems in different partitions. However, conventional operating systems may need to be modified in some aspects in order to make them compatible with the inventive system, depending on how the system is configured. Some sample modifications for the illustrative embodiment are listed below:

Detailed Description Text (79):
3. An instance may need to be capable of starting at an arbitrary physical address and may not be able to reserve any specific physical address in order to avoid conflicting with other operating systems running at that particular address.

Detailed Description Text (82):
6. An instance may need to be capable of running entirely within its "private memory" if it is used in a system where instances do not share memory. Alternatively, an instance may need to be capable of using physical "shared memory" for communicating or sharing data with other instances running within the computer if the instance is part of a system in which memory is shared. In such a shared memory system, an instance may need to be capable of mapping physical "shared memory" as identified in the configuration tree into its virtual address space, and the virtual address spaces of the "processes" running within that operating system instance.

Detailed Description Text (86):
Each console program provides a number of callback functions to allow the associated operating system instance to change the configuration of the APMP system, for example, by creating a new community or partition, or altering the ownership of memory fragments. In addition, other callback functions provide the ability to remove a community, or partition, or to start operation on a newly-created partition.

Detailed Description Text (87):
However, callback functions do not cause any changes to take place on the running operating system instances. Any changes made to the configuration tree must be acted upon by each instance affected by the change. The type of action that must take place in an instance when the configuration tree is altered is a function of the type of change, and the operating system instance capabilities. For example, moving an input/output processor from one partition to another may require both partitions to reboot. Changing the memory allocation of fragments, on the other hand, might be handled by an operating system instance without the need for a reboot.

Detailed Description Text (94):
Finally, an operating system instance is booted in at least one of the partitions as indicated in step 510. The first operating system instance to boot creates an APMP database and fills in the entries as described below. APMP databases store information relating to the state of active operating system instances in the system. The routine then finishes in step 512. It should be noted that an instance is not required to participate in an APMP system. The instance can choose not to participate or to participate at a time that occurs well after boot. Those instances which do participate form a "sharing set." The first instance which decides to join a sharing set must create it there can be multiple sharing sets operating on a

single APMP system and each sharing set has its own APMP database.

Detailed Description Text (96):
An operating system instance running on a platform which is also running the APMP computer system does not necessarily have to be a member of the APMP computer system. The instance can attempt to become a member of the APMP system at any time after booting. This may occur either automatically at boot, or after an operator-command explicitly initiates joining. After the operating system is loaded at boot time, the operating system initialization routine is invoked and examines a stored parameter to see whether it specifies immediate joining and, if so, the system executes a joining routine which is part of the APMP computer system. An operator command would result in an execution of the same routine.

Detailed Description Text (98):
An important data structure supporting the inventive software allocation of resources is the APMP database which keeps track of operating system instances which are members of a sharing set. The first operating system instance attempting to set up the APMP computer system initializes an APMP database, thus creating, or instantiating, the inventive software resource allocations for the initial sharing set. Later instances wishing to become part of the sharing set join by registering in the APMP database associated with that sharing set. The APMP database is a shared data structure containing the centralized information required for the management of shared resources of the sharing set. An APMP database is also initialized when the APMP computer system is reformed in response to an unrecoverable error.

Detailed Description Text (100):
The initial, header portion of an APMP database is the first part of the APMP database mapped by a joining operating system instance. Portions of the header are accessed before the instance has joined the sharing set, and, in fact, before the instance knows that the APMP computer system exists.

Detailed Description Text (111):
An APMP database is stored in shared memory. The initial fixed portion of N physically contiguous pages occupies the first N pages of one of two memory ranges allocated by the first instance to join during initial partitioning of the hardware. The instance directs the console to store the starting physical addresses of these ranges in the configuration tree. The purpose of allocating two ranges is to permit fail over in case of hardware memory failure. Memory management is responsible for mapping the physical memory into virtual address space for the APMP database.

Detailed Description Text (112):
The detailed actions taken by an operating system instance are illustrated in FIG. 6. More specifically, when an operating system instance wishes to become a member of a sharing set, it must be prepared to create the APMP computer system if it is the first instance attempting to "join" a non-existent system. In order for the instance to determine whether an APMP system already exists, the instance must be able to examine the state of shared memory as described above. Further, it must be able to synchronize with other instances which may be attempting to join the APMP system and the sharing set at the same time to prevent conflicting creation attempts. The master console creates the configuration tree as discussed above. Subsequently, a region of memory is initialized by the first, or primary, operating system instance to boot, and this memory region can be used for an APMP database.

Detailed Description Text (114):
The goal of the initial actions taken by all operating system instances is to map the header portion of the APMP database and initialize primitive inter-instance interrupt handling to lay the groundwork for a create or join decision. The routine used is illustrated in FIG. 6 which begins in step 600. The first action taken by each instance (step 602) is to engage memory management to map the initial segment of the APMP database as described above. At this time, the array of node blocks in the second database section is also mapped. Memory management maps the initial and second segments of the APMP database into the primary operating system address space and returns the start address and length. The instance then informs the console to store the location and size of the segments in the configuration tree.

Detailed Description Text (115):
Next, in step 604, the initial virtual address of the APMP database is used to allow the initialization routine to zero interrupt reason masks in the node block assigned to the current instance.

Detailed Description Text (117):
Next, in step 608, the virtual address (VA) of the APMP database is stored in a private cell which is examined by an inter-processor interrupt handler. The handler examines this cell to determine whether to test the per-instance interrupt reason mask in the APMP database header for work to do. If this cell is zero, the APMP database is not mapped and nothing further is done by the handler. As previously discussed, the entire APMP database, including this mask, is initialized so that the handier does nothing before the address is stored. In addition, a clock interrupt handler can examine the same private cell to determine whether to increment the instance-specific heartbeat field for this instance in the appropriate node block. If the private cell is zero, the interrupt handler does not increment the heartbeat field.

Detailed Description Text (130):
If a member instance is judged dead, or disinterested, and it has not notified the APMP computer system of its intent to shut down or crash, the instance is removed from the APMP system. This may be done, for example, by setting the "bugcheck" bit in the instance primitive interrupt mask and sending an IP interrupt to all CPU's of the instance. As a rule, shared memory may only be accessed below the hardware priority of the IP interrupt. This insures that if the CPUs in the instance should attempt to execute at a priority below that of the IP interrupt, the IP interrupt will occur first and thus the CPU will see the "bugcheck" bit before any lower priority threads can execute. This insures the operating system instance will crash and not touch shared resources such as memory which may have been reallocated for other purposes when the instances were judged dead. As an additional or alternative mechanism, a console callback (should one exist) can be invoked to remove the instance. In addition, in accordance with a preferred embodiment, whenever an instance disappears or drops out of the APMP computer system without warning, the remaining instances perform some sanity checks to determine whether they can continue. These checks include verifying that all pages in the APMP database are still accessible, i.e. that there was not a memory failure.

Detailed Description Text (133):
CPU ownership is indicated in a number of ways, in a number of structures dictated by the entity that is managing the resource at the time. In the most basic case, the CPU can be in an unassigned state, available to all partitions that reside in the same sharing set as the CPU. Eventually that CPU is assigned to a specific partition, which may or may not be running an operating system instance. In either case, the partition reflects its ownership to all other partitions through the configuration tree structure, and to all operating system instances that may run in that partition through the AVAILABLE bit in the HWRPB per-CPU flags field.

Detailed Description Text (134):
If the owning partition has no operating system instance running on it, its console is responsible for responding to, and initiating, transition events on the resources within it. The console decides if the resource is in a state that allows it to migrate to another partition or to revert back to the unassigned state.

Detailed Description Text (135):
If, however, there is an instance currently running in the partition, the console relinquishes responsibility for initiating resource transitions and is responsible for notifying the running primary of the instance when a configuration change has taken place. It is still the facilitator of the underlying hardware transition, but control of resource transitions is elevated one level up to the operating system instance. The transfer of responsibility takes place when the primary CPU executes its first instruction outside of console mode in a system boot.

Detailed Description Text (136):
Operating system instances can maintain ownership state information in any number of ways that promote the most efficient usage of the information internally. For

example, a hierarchy of state bit vectors can be used which reflect the instance-specific information both internally and globally (to other members sharing an APMP database).

Detailed Description Text (137):
The internal representations are strictly for the use of the instance. They are built up at boot time from the underlying configuration tree and HWRPB information, but are maintained as strict software constructs for the life of the operating system instance. They represent the software view of the partition resources available to the instance, and may--through software rule sets--further restrict the configuration to a subset of that indicated by the physical constructs. Nevertheless, all resources in the partition are owned and managed by the instance--using the console mechanisms to direct state transitions--until that operating system invocation is no longer a viable entity. That state is indicated by halting the primary CPU once again back into console mode with no possibility of returning without a reboot.

Detailed Description Text (140):
During runtime, the current_owner field reflects the partition where a CPU is executing. The AVAILABLE bit in the per-CPU flags field in the HWRPB remains the ultimate indicator of whether a CPU is actually available, or executing, for SMP operation with an operating system instance, and has the same meaning as in conventional SMP systems.

Detailed Description Text (142):
Using the above-described computer system, various system configurations can be achieved simply by appropriately manipulating the configuration tree. FIG. 9 is an overview which shows what the architectural description looks like when the inventive computer system is instantiated on a machine. This example is a nine CPU SMP machine 900 that has been partitioned into three partitions, 901, 902 and 904. Each partition has one or more CPUs (groups 906, 908 and 910) and some private I/O resources (designated as 912, 914 and 916) and is running an instance of an operating system (illustrated schematically as 918, 920 and 922). The operating systems may be three copies of the same operating system or three different operating systems. The memory in this system is cache-coherent shared memory. The common memory 924 has been partitioned into three segments of private memory, 926, 928 and 930, one for each operating system instance, and the remainder 932 is shared memory for all three operating system instances to use cooperatively.

Detailed Description Text (143):
A configuration such as that illustrated schematically in FIG. 9 can be created by appropriate entries in the configuration tree illustrated in FIGS. 3 and 4. What has been created is the equivalent of three physically independent computers within a single computer. This construction is the basis for one embodiment of the inventive computer system--multiple independent instances of operating systems executing cooperatively in a single computer. The inventive computer system is a set of coordinated heterogeneous operating systems, in a single computer, communicating via shared memory. An instance of the operating system in such a system can be clustered with other operating system instances within the same computer system or with operating system instances in other computer systems.

Detailed Description Text (144):
Any instance on an operating system in the inventive computer system can be an SMP configuration. The number of CPUs is part of the definition of an instance and is determined by the configuration tree entries. Because an instance in the inventive system is a complete operating system, all applications behave the same as they would on a traditional, single-instance computer. For example, existing single-system applications will run without changes on instances in a the inventive computer system. Existing cluster applications will also run without changes on clustered instances in an operating system within the system. The system is more available than the traditional single-system-view SMP system because multiple instances of operating systems are running in the system. Consequently, if a single instance fails, other instances can continue to run in spite of the hardware or software error that caused the failure.

Detailed Description Text (145):
In such a system, memory 924 is logically partitioned into private (926, 928 and 930) and shared sections (932). As a minimum, each operating system instance has its own private memory section. No other instance can map into this physical memory section, except under direction of the owning operating system instance. Some of the shared memory 924 is available for instances of operating systems (918, 920 and 912) to communicate with one another, and the rest of the shared memory is available for applications.

Detailed Description Text (147):
Although there are three separate operating system instances in FIG. 9, there is an expectation of cooperation between the instances. Such cooperation can be defined in three broad categories or computing models:

Detailed Description Text (148):
1) Shared Nothing--the operating system instances do not share any resources but agree to cooperate by not interfering with each other.

Detailed Description Text (149):
2) Shared Partial (shared something)--the operating system instances agree to cooperate and share some limited resources, such as memory or storage.

Detailed Description Text (150):
3) Shared Everything--the operating system instances agree to cooperate fully and share all resources available, to the point where the operating system instances present a single cohesive entity to the network. For example, the operating system instances may appear to the user as an OpenVMS Cluster.

Detailed Description Text (151):
Even though some operating system instances are sharing resources, one or more operating system instances can execute in total software isolation from all others. An instance that exists without sharing any resources is called an independent instance and does not participate at all in shared memory use. More particularly, neither the base operating system, nor its applications, access shared memory. The inventive computer system could consist solely of independent instances; such a system would resemble traditional mainframe style partitioning.

Detailed Description Text (152):
FIG. 10 shows how the inventive system can be configured to support a "shared nothing" model of computing. In this example, three partitions, 1006, 1010 and 1014, have been created within a single machine 1000, each running an instance of an operating system. The available twelve CPUs have been split arbitrarily equally between the partitions in this example, 1006,1010 and 1014, as illustrated. The available memory has been divided into private memory and assigned to the instances. In FIG. 10, private memory divisions 1016, 1018 and 1020 are illustrated. Both code and data for each instance is stored in the the private memory assigned to that instance. Although the memory, 1016, 1018 and 1020 has been illustrated in FIG. 10 as divided equally, the inventive architecture supports arbitrary division of memory between instances. Thus, if a partition has large memory requirements, and another that has limited memory needs, the system can accommodate both, in order to maximize use of available memory.

Detailed Description Text (156):
The advantage of the configuration illustrated in FIG. 11 is that large shared cache memories (for example, databases or file systems) can now be created and used jointly by several instances. The system also has the advantage that an instance of an operating system can deactivate or leave the configuration, and upon rejoining, can re-map into a still active cache memory. As cache memories become larger, this ability to remap into an existing memory is extremely important since it is very time-consuming to load all entries in very large cache memories into a private memory space.

Detailed Description Text (162):
Regardless of the computing model, or combination of models, that are run under the inventive architecture, the ability to dynamically re-allocate resources is possible

without affecting the integrity of any of the operating system instances. Specifically, the ability to migrate CPUs and memory between partitions is supported. Thus, the system offers the possibility of a more linear scaling of system resources because system managers can assign resources to match application requirements as business needs grow or change. When a CPU is added to an inventive computer configuration, it can be assigned to any instance of operating system and reassigned later during system operation so that a "trial and error" method of assigning resources is a viable strategy. In particular, system managers can migrate CPUs among instances of operating systems until the most effective combination of resources is found. All instances of operating systems and their applications continue to run as CPUs are migrated. Finally, the distribution of interrupts across instances provides many I/O configuration possibilities; for example, a system's I/O workload can be partitioned so that certain I/O traffic is done on specific instances.

Detailed Description Text (164):
The inventive system allows individual I/O subsystems to be quiesced by nature of the fact that an operating system instance running in a partition can be shut down without impacting the remaining partitions. In general, if the hardware continues to run except for the component being swapped, the software keeps as many of the instances and their applications as possible running. The invention also supports "hot inswapping" which is a hardware feature that allows resources to be added to the system while the system is operating and electrical power is applied. An example of hot inswapping is additional memory. Assuming an operating system instance on a particular partition is capable of dynamically mapping additional memory, the system can absorb hot inswapped memory into the active computing environments running in the machine.

Detailed Description Text (165):
A software implementation of the above-described embodiment may comprise a series of computer instructions either fixed on a tangible medium, such as a computer readable media, e.g. a diskette, a CD-ROM, a ROM memory, or a fixed disk, or transmissible to a computer system, via a modem or other interface device, over a medium. The medium can be either a tangible medium, including but not limited to, optical or analog communications lines, or may be implemented with wireless techniques, including but not limited to microwave, infrared or other transmission techniques. It may also be the Internet. The series of computer instructions embodies all or part of the functionality previously described herein with respect to the invention. Those skilled in the art will appreciate that such computer instructions can be written in a number of programming languages for use with many computer architectures or operating systems. Further, such instructions may be stored using any memory technology, present or future, including, but not limited to, semiconductor, magnetic, optical or other memory devices, or transmitted using any communications technology, present or future, including but not limited to optical, infrared, microwave, or other transmission technologies. It is contemplated that such a computer program product may be distributed as a removable media with accompanying printed or electronic documentation, e.g., shrink wrapped software, pre-loaded with a computer system, e.g., on system ROM or fixed disk, or distributed from a server or electronic bulletin board over a network, e.g., the Internet or World Wide Web.

Detailed Description Text (166):
Although an exemplary embodiment of the invention has been disclosed, it will be apparent to those skilled in the art that various changes and modifications can be made which will achieve some of the advantages of the invention without departing from the spirit and scope of the invention. For example, it will be obvious to those reasonably skilled in the art that, although the description was directed to a particular hardware system and operating system, other hardware and operating system software could be used in the same manner as that described. Other aspects, such as the specific instructions utilized to achieve a particular function, as well as other modifications to the inventive concept are intended to be covered by the appended claims.

Detailed Description Paragraph Table (9):
Step 702 the creator instance sets the APMP system state and its node block state to "initializing." Step 704 the creator instance calls a size routine for each system

service with the address of its length field in the header. Step 706 the resulting length fields are summed and the creator instance calls memory management to allocate space for the entire APMP database by creating a new mapping and deleting the old mapping. Step 708 the creator instance fills in the offsets to the beginnings of each system service segment. Step 710 the initialization routine for each service is called with the virtual addresses of the APMP database, the service segment and the segment length. Step 712 the creator instance initializes a membership mask to make itself the sole member and increments an incarnation count. It then sets creation time, software version, and other creation parameters. Step 714 the instance then sets itself as its own big and little brother (for heartbeat monitoring purposes as described below). Step 716 the instance then fills in its instance state as "member" and the APMP system state as "operational." Step 718 finally, the instance releases the APMP database lock. The routine then ends in step 720.

Detailed Description Paragraph Table (10):
Step 802 the instance checks to make sure that its instance name is unique. If another current member has the instance's proposed name, joining is aborted. Step 804 the instance sets the APMP system state and its node block state to "instance joining" Step 806 the instance calls a memory management routine to map the variable portion of the APMP database into its local address space. Step 808 the instance calls system joining routines for each system service with the virtual addresses of the APMP database and its segment and its segment length. Step 810 if all system service joining routines report success, then the instance joining routine continues. If any system service join routine fails, the instance joining process must start over and possibly create a new APMP computer system. Step 812 assuming that success was achieved in step 810, the instance adds itself to the system membership mask. Step 814 the instance selects a big brother to monitor its instance health as set forth below. Step 816 the instance fills in its instance state as "member" and sets a local membership flag. Step 818 the instance releases the configuration database lock. The routine then ends in step 820.

Other Reference Publication (1):
Beck, "AAMP: A Multiprocessor Approach for Operating System and Application Migration," Operating Systems Review 24:41-55 (Apr. 1990).

Other Reference Publication (3):
Ohmori et al., "System management of MICS-II--a virtual machine complex"; Third USA-Japan Computer Conference Proceedings, San Francisco, Calif., Sep. 1978, pp. 425-429.

Other Reference Publication (4):
Beck, "AAMP: A Multiprocessor Approach for Operating System and Application Migration," Operating Systems Review 24:41-55 (1990).

Other Reference Publication (7):
Rashid et al., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," IEEE Transactions on Computers 37:896-907 (1988).

Other Reference Publication (18):
J. Chapin, et al., Hive: Fault Containment for Shared-Memory Multiprocessors, The 15th ACM Symposium on Operating Systems Principles, 12/95.

Other Reference Publication (20):
Rohit Chandra, et al., Scheduling and Page Migration for Multiprocessor Compute Servers, Sixth International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS-V1), 10/94.

Other Reference Publication (33):
C. Hanna, Logical Partitioning Methodologies, CMG (Conference), 1993.

CLAIMS:

1. A computer system having a plurality of configurable system resources including

processors, memory, and I/O circuitry different from the memory, the computer system comprising: an interconnection mechanism which electrically interconnects hardware components including the processors, memory and I/O circuitry so that each processor has electrical access to all of the memory and at least some of the I/O circuitry, such that the memory is configurable to allow data access by any one or more of the processors; and a software mechanism that divides at least a portion of the plurality of system resources into a plurality of soft partitions, each soft partition including at least one processor, some private memory and some private I/O circuitry, with at least one operating system instance running in each soft partition; wherein a first pair of operating system instances share a first subset of the plurality of system resources as a first sharing set, and wherein a second pair of operating system instances share a second subset of the plurality of system resources different from the first subset of operating system resources as a second sharing set; and wherein each of said hardware components has an associated affinity handle which specifies a configuration that is more optimal than other possible configurations.

2. A computer system according to claim 1 wherein all of the memory is dedicated to the first pair of operating system instances and to the second pair of operating system instances.

3. A computer system according to claim 1 wherein at least some memory is dedicated to each operating system instance in the first pair of operating system instances and at least some memory is dedicated to each operating system instance in the second pair of operating system instances.

5. A computer system according to claim 3 wherein at least some of the memory is shared among the first pair of operating system instances and the second pair of operating system instances.

16. A computer system according to claim 1, wherein the first pair of operating system instances sharing the first sharing set are comprised in a different soft partition from the second pair of operating system instances sharing the second sharing set.

17. A method for operating a computer system having a plurality of configurable system resources including processors, memory, and I/O circuitry different from the memory, the method comprising the steps of: (a) electrically interconnecting hardware components including the processors, memory and I/O circuitry so that each processor has electrical access to all of the memory and at least some of the I/O circuitry, such that the memory is configurable to allow data access by any one or more of the processors; (b) forming a plurality of soft partitions, each soft partition including at least one processor, some private memory and some private I/O circuitry; (c) running at least one operating system instance in each soft partition; (d) creating a first pair of operating system instances which share a first subset of the plurality of system resources as a first sharing set; and (e) creating a second pair of operating system instances which share a second subset of the plurality of system resources different from the first subset of operating system resources as a second sharing set; (f) associating an affinity handle with each hardware component, said affinity handle specifies a configuration that is more optimal than other possible configurations.

18. A method according to claim 17 further comprising the step of: (f) dedicating all of the memory to the first pair of operating system instances and to the second pair of operating system instances.

19. A method according to claim 17 further comprising the steps of: (g) dedicating at least some memory to each operating system instance in the first pair of operating system instances; and (h) dedicating at least some memory to each operating system instance in the second pair of operating system instances.

21. A method according to claim 19 further comprising the step of: (j) sharing at least some of the memory among the first pair of operating system instances and the second pair of operating system instances.

23. A method according to claim 21 wherein step (j) comprises the step of: (j1) sharing a cache-coherent memory among the first pair of operating system instances and the second pair of operating system instances.

24. A method according to claim 23 wherein step (j1) comprises the step of: (j1a) sharing a database cache among the first pair of operating system instances and the second pair of operating system instances.

25. A method according to claim 23 wherein step (j1) comprises the step of: (j1b) sharing a file system cache among the first pair of operating system instances and the second pair of operating system instances.

26. A method according to claim 23 wherein step (j1) comprises the step of: (j1c) sharing a lock cache among the first pair of operating system instances and the second pair of operating system instances.

27. A method according to claim 23 wherein step (j1) comprises the step of: (j1d) sharing a cluster interconnect among the first pair of operating system instances and the second pair of operating system instances.

32. A method according to claim 17, wherein step (e) further comprises: (e1) creating the second pair of operating system instances in a different software partition from the first pair of operating system instances.

33. A computer program product for operating a computer system having a plurality of configurable system resources including hardware components comprising processors, memory, and I/O circuitry different from the memory, and a mechanism for electrically interconnecting the processors, memory and I/O circuitry so that each processor has electrical access to all of the memory and at least some of the I/O circuitry, such that the memory is configurable to allow data access by any one or more of the processors, the computer program product comprising a computer usable medium having computer readable program code thereon including: program code for forming a plurality of soft partitions, each soft partition including at least one processor, some private memory and some private I/O circuitry; program code for running at least one operating system instance in each soft partition; program code for creating a first pair of operating system instances which share a first subset of the plurality of system resources as a first sharing set; program code for creating a second pair of operating system instances which share a second subset of the plurality of system resources different from the first subset of operating system resources as a second sharing set; and program code for associating an affinity handle with each hardware component, said affinity handle specifies a configuration that is more optimal than other possible configurations.

34. A computer program product according to claim 33 further comprising program code for dedicating all of the memory to the first pair of operating system instances and to the second pair of operating system instances.

35. A computer program product according to claim 33 further comprising: program code for dedicating at least some memory to each operating system instance in the first pair of operating system instances; and program code for dedicating at least some memory to each operating system instance in the second pair of operating system instances.

37. A computer program product according to claim 35 further comprising program code for sharing at least some of the memory among the first pair of operating system instances and the second pair of operating system instances.

39. A computer program product according to claim 37 wherein the program code for sharing at least some of the memory comprises program code for sharing a cache-coherent memory among the first pair of operating system instances and the second pair of operating system instances.

40. A computer program product method according to claim 39 wherein the program code for sharing cache-coherent memory comprises program code for sharing a database cache among the first pair of operating system instances and the second pair of

operating system instances.

41. A computer program product according to claim 39 wherein the program code for sharing cache-coherent memory comprises program code for sharing a file system cache among the first pair of operating system instances and the second pair of operating system instances.

42. A computer program product according to claim 39 wherein the program code for sharing cache-coherent memory comprises program code for sharing a lock cache among the first pair of operating system instances and the second pair of operating system instances.

43. A computer program product according to claim 39 wherein the program code for sharing cache-coherent memory comprises program code for sharing a cluster interconnect among the first pair of operating system instances and the second a pair of operating system instances.

48. A computer program product according to claim 33, wherein the program code for creating the second pair of operating system instances further comprises program code for creating the second pair of operating system instances in a different soft partition from the first pair of operating system instances.

First Hit Fwd Refs
End of Result Set

☐ Generate Collection Print

L21: Entry 1 of 1 File: USPT Jun 7, 1994

DOCUMENT-IDENTIFIER: US 5319760 A
TITLE: Translation buffer for virtual machines with address space match

Detailed Description Text (28):
Typically, in a multitasking system, several processes may reside in physical memory 12 (or caches) at the same time, so memory protection and multiple address spaces (using address space numbers) are used by the CPU 10 to ensure that one process will not interfere with either other processes or the operating system. To further improve software reliability, four hierarchical access modes (privilege modes) provide memory access control. They are, from most to least privileged: kernel, executive, supervisor, and user, referring to operating system modes and application programs. Protection is specified at the individual page level, where a page may be inaccessible, read-only, or read/write for each of the four access modes. Accessible pages can be restricted to have only data or instruction access.

Detailed Description Text (34):
The CPU 10 generates memory references by first forming a virtual address 76 on bus 56, representing the address within the entire virtual range 97 as seen in FIG. 11, defined by the 43-bit address width referred to, or that portion of the address width used by the operating system. Then using the page tables 85, 88, 90 in memory, or the translation buffer 36 or 48, the virtual address is translated to a physical address represented by an address map 98; the physical memory is constrained by the size of the main memory 12. The translation is done for each page (e.g., an 8 Kbyte block), so a virtual page address for a page 99 in the virtual memory map 97 is translated to a physical address 99' for a page (referred to as a page frame) in the physical memory map 98. The page tables are maintained in memory 12 or cache 20 to provide the translation between virtual address and physical address, and the translation buffer is included in the CPU to hold the most recently used translations so a reference to the page tables in memory 12 need not be made in most cases to obtain the translation before a data reference can be made; the time needed to make the reference to a page table in memory 12 would far exceed the time needed to obtain the translation from the translation buffer. Only the pages used by tasks currently executing (and the operating system itself) are likely to be in the physical memory 12 at a given time; a translation to an address 99' is in the page table 85, 88, 90 for only those pages actually present in physical memory 12. When the page being referenced by the CPU 10 is found not to be in the physical memory 12, a page fault is executed to initiate a swap operation in which a page from the physical memory 12 is swapped with the desired page maintained in the disk memory 13, this swap being under control of the operating system. Some pages in physical memory 12 used by the operating system kernel, for example, or the page tables 85, 88, 90 themselves, are in fixed positions and may not be swapped to disk 13 or moved to other page translations; most pages used by executing tasks, however, may be moved freely within the physical memory 12 by merely keeping the page tables updated.

Detailed Description Text (36):
Referring to FIG. 12, the context block 100 contains four stack pointers in fields 101, these being stack pointers for the kernel, the executive, the supervisor and

h e b b g e e e f c e g e ge

the user. The page table base register 86 is in field 102. The address space number for this process (to be loaded to register 95) is in field 103. Other fields 104 are for values not material here. The location of this block 100 in memory is specified for the current process by a context block base register 105. A swap context instruction saves the privileged context of the current process into the context block specified by this register 105, loads a new value into the register 105, and then loads the privileged context of the new process from the new block 100 into the appropriate hardware registers 43, etc.

Detailed Description Text (41):
The purpose of the invention is to maximize system performance, while allowing the virtual machines to run in kernel mode so they can execute privileged instructions (otherwise they would have to use traps for these operations, at considerable performance penalty); to do this the VMM must constrain the VMs. The problem addressed in implementing this invention is to keep the address spaces of the several VMs and the VMM itself separate from each other, while at the same time maximizing system performance by providing the match function in the TB to the VMs. A further constraint on the solution is that it is expected that the VMs and the VMM will use the same virtual addresses for different purposes. Thus, it is not a solution to allocate separate address regions to the VMM from those allocated to the VMs.

Detailed Description Paragraph Table (4):
TABLE A Page Table Entry Fields in the page table entry are interpreted as follows: Bits Description <0> Valid (V) - Indicates the validity of the PFN field. <1> Fault On Read (FOR) - When set, a Fault On Read exception occurs on an attempt to read any location in the page. <2> Fault On Write (FOW) - When set, a Fault On Write exception occurs on an attempt to write any location in the page. <3> Fault on Execute (FOE) - When set, a Fault On Execute exception occurs on an attempt to execute an instruction in the page. <4> Address Space Match (ASM) - When set, this PTE matches all Address Space Numbers. For a given VA, ASM must be set consistently in all processes. <6:5> Granularity hint (GH) - Software may set these bits to a non-zero value to supply a hint to the translation buffer that a block of pages can be treated as a single larger page. <7> Reserved for future use. <8> Kernel Read Enable (KRE) - This bit enables reads from kernel mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in kernel mode, an Access Violation occurs. This bit is valid even when V = 0. <9> Executive Read Enable (ERE) - This bit enables reads from executive mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in executive mode, an Access Violation occurs. This bit is valid even when V = 0. <10> Supervisor Read Enable (SRE) - This bit enables reads from supervisor mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in supervisor mode, an Access Violation occurs. This bit is valid even when V = 0. <11> User Read Enable (URE) - This bit enables reads from user mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in user mode, an Access Violation occurs. This bit is valid even when V = 0. <12> Kernel Write Enable (KWE) - This bit enables writes from kernel mode. If this bit is a 0 and a STORE is attempted while in kernel mode, an Access Violation occurs. This bit is valid even when V = 0. <13> Executive Write Enable (EWE) - The bit enables writes from executive mode. If this bit is a 0 and a STORE is attempted while in executive mode, an Access Violation occurs. <14> Supervisor Write Enable (SWE) - This bit enables writes from supervisor mode. If this bit is a 0 and a STORE is attempted while in executive mode, an Access Violation occurs. <15> User Write Enable (UWE) - This bit enables writes from user mode. If this bit is a 0 and a STORE is attempted while in user mode, Access Violation occurs. <31:16> Reserved for software. <63:32> Page Frame Number (PFN) - The PFN field always points to a page boundary. If V is set, the PFN is concatenated with the Byte Within Page bits of the virtual address to obtain the physical address. If V is clear, this field may be used by software.

h e b b g e e e f c e g e ge

First Hit Fwd Refs
End of Result Set

☐ Generate Collection Print

L19: Entry 1 of 1 File: USPT Jun 7, 1994

DOCUMENT-IDENTIFIER: US 5319760 A
TITLE: Translation buffer for virtual machines with address space match

Abstract Text (1):
A central processing unit (CPU) executing a virtual memory management system employs a translation buffer for caching recently used page table entries. When more than one process is executing on the CPU, the translation buffer is usually flushed when a context switch is made, even though some of the entries would still be valid for commonly-referenced memory areas. An address space number feature is employed to allow entries to remain in the translation buffer for processes not currently executing, and the separate processes or the operating system can reuse entries in the translation buffer for such pages of memory that are commonly referenced. To allow this, an "address space match" entry in the page table entry signals that the translation buffer content can be used when the address tag matches, even though the address space numbers do not necessarily match. When executing virtual machines on this CPU, with a virtual machine monitor, the address space match feature is employed among processes of a virtual machine, but an additional entry is provided to disable the address space match feature for all address space numbers for the virtual machine monitor.

Brief Summary Text (3):
Ser. No. 547,630, filed Jun. 29, 1990, entitled IMPROVING PERFORMANCE IN REDUCED INSTRUCTION SET PROCESSOR, by Richard L. Sites and Richard T. Witek, inventors;

Brief Summary Text (6):
In the above-identified copending application Ser. No. 547,630, a reduced instruction set processor chip is shown which implements a virtual memory management system. A virtual address is translated to a physical address in such a system before a memory reference is executed, where the physical address is that used to access the main memory. The physical addresses are maintained in a page table, indexed by the virtual address, so whenever a virtual address is presented the memory management system finds the physical address by referencing the page table. At a given time, a process executing on such a machine will probably be using only a few pages, and so these most-likely used page table entries are kept in a translation buffer within the CPU chip itself, eliminating the need to make a memory reference to fetch the page table entry.

Brief Summary Text (9):
A mechanism using so-called "address space numbers" is implemented in a processor to reduce the need for invalidation of cached address translations in the translation buffer for process-specific addresses when a context switch occurs. The address space number (process tag) for the current process is loaded to a register in the processor to become part of the current state; this loading is done by a privileged instruction from a process-specific block in memory. Thus, each process has associated with it an address space number, which is an arbitrarily-assigned number generated by the operating system. This address space number is maintained as part of the machine state, and also stored in the translation buffer for each page entry belonging to that process. When a memory reference is made, as part of

the tag match in the translation buffer, the current address space number is compared with the entry in the translation buffer, to see if there is a match. To accommodate sharing of entries, an address space match function can be added to the comparison; a "match" bit in the entry can turn on or off the requirement for matching address space numbers. If turned on, the entry will "match" if the address tags match, regardless of the address space numbers. The operating system can thus load certain page table entries with this match bit on so these pages are shared by all processes.

Brief Summary Text (11):
The several virtual machines and the virtual machine monitor running on a CPU must have their memory spaces kept separate and isolated from one another, but yet maximize system performance. To this end, the virtual machines and the virtual machine monitor must be able to use the same virtual addresses for different purposes. However, when context switching from one virtual machine to another, or to or from the virtual machine monitor, needlessly flushing entries in the translation buffer which will be used in the new context imposes a performance penalty. Therefore it is important to offer both address space numbers and the match feature when implementing virtual machines.

Brief Summary Text (13):
In accordance with one embodiment of the invention, a CPU executing a virtual memory management system employs an address space number feature to allow entries to remain in the translation buffer for processes not currently executing, and the separate processes or the operating system can reuse entries in the translation buffer for such pages of memory that are commonly referenced. To allow this, an "address space match" entry in the page table entry signals that the translation buffer content can be used when the address tag matches, even though the address space numbers do not necessarily match. When executing virtual machines on this CPU, with a virtual machine monitor, the address space match feature is employed within a virtual machine, but an additional entry is provided to disable the address space match feature for all address space numbers for the virtual machine monitor. In another embodiment, an additional entry is provided in the translation buffer to restrict the address space match feature to those address spaces associated with a single virtual machine or virtual machine monitor.

Drawing Description Text (4):
FIG. 2 is an electrical diagram in block form of the instruction unit or I-box of the CPU of FIG. 1;

Drawing Description Text (6):
FIG. 4 is an electrical diagram in block form of the addressing unit or A-box in the CPU of FIG. 1;

Drawing Description Text (9):
FIG. 7 is a diagram of the instruction formats used in the instruction set of the CPU of FIGS. 1-5;

Drawing Description Text (10):
FIG. 8 is a diagram of the format of a virtual address used in the CPU of FIGS. 1-5;

Drawing Description Text (12):
FIG. 10 is a diagram of the address translation mechanism used in the CPU of FIGS. 1-5;

Drawing Description Text (13):
FIG. 11 is a diagram of the virtual-to-physical address mapping used in the system of FIGS. 1-5;

Drawing Description Text (15):
FIG. 13 is a table of address space numbers for an example of operating the system of FIGS. 1-5 with virtual machines;

Detailed Description Text (3):
The CPU 10 is of a single-chip integrated circuit device, in an example embodiment, although features of the invention could be employed as well in a processor constructed of integrated circuit devices mounted on boards. Within the single chip an integer execution unit 16 (referred to as the "E-box") is included, along with a floating point execution unit 17 (referred to as the "F-box"). Instruction fetch and decoding is performed in an instruction unit 18 or "I-box". An address unit or "A-box" 19 performs the functions of address generation, memory management, write buffering and bus interface; the virtual address system with translation buffer according to the invention is implemented in the address unit 19. The memory is hierarchical, with on-chip instruction and data caches being included in the instruction unit 18 and address unit 19 in one embodiment, while a larger, second-level cache 20 is provided off-chip, being controlled by a cache controller in the address unit 19.

Detailed Description Text (4):
The CPU 10 employs an instruction set as described in application Ser. No. 547,630, in which all instructions are of a fixed size, in this case 32-bit or one longword. Memory references are generally aligned quadwords, although integer data types of byte, word, longword and quadword are handled internally. As used herein, a byte is 8-bits, a word is 16-bits or two bytes, a longword is 32-bits or four bytes, and a quadword is 64-bits or eight bytes. The data paths and registers within the CPU 10 are generally 64-bit or quadword size, and the memory 12 and caches use the quadword as the basic unit of transfer. Performance is enhanced by allowing only quadword or longword loads and stores.

Detailed Description Text (5):
Referring to FIG. 2, the instruction unit 18 or I-box is shown in more detail. The primary function of the instruction unit 18 is to issue instructions to the E-box 16, A-box 19 and F-box 17. The instruction unit 18 includes an instruction cache 21 which stores perhaps 8 Kbytes of instruction stream data, and this instruction stream data is loaded to an instruction register 22 in each cycle for decoding. In one embodiment, two instructions are decoded in parallel. An instruction is decoded in a decoder 23, producing register addresses on lines 26 and control bits on microcontrol bus 28 to the appropriate elements in the CPU 10.

Detailed Description Text (6):
The instruction unit 18 contains address generation circuitry 29, including a branch prediction circuit 30 responsive to the instructions in the instruction stream to be loaded into register 22. The prediction circuit 30 is used to predict branch addresses and to cause address generating circuitry 29 to prefetch the instruction stream before needed. The virtual PC (program counter) 33 is included in the address generation circuitry 29 to produce addresses for instruction stream data in the selected order.

Detailed Description Text (7):
The instruction unit 18 contains a fully associative translation buffer (TB) 36 to cache recently used instruction-stream address translations and protection information for 8 Kbyte pages. Although 64-bit addresses are nominally possible, as a practical matter 43-bit addresses are adequate. Every cycle the 43-bit virtual program counter 33 is presented to the instruction stream TB 36. If the page table entry (PTE) associated with the virtual address from the virtual PC is cached in the TB 36 then the page frame number (PFN) and protection bits for the page which contains the virtual PC are used by the instruction unit 18 to complete the address translation and access checks. A physical address is thus applied to the address input 37 of the instruction cache 21, or if there is a cache miss then this

instruction stream physical address is applied by the bus 38 through the address unit 19 to the cache 20 or memory 12.

Detailed Description Text (8):
The execution unit or E-box 16 is shown in more detail in FIG. 3. The execution unit 16 contains the 64-bit integer execution datapath including an arithmetic/logic unit (ALU) 40, a barrel shifter 41, and an integer multiplier 42. The execution unit 16 also contains the 32-register 64-bit wide register file 43, containing registers R0 to R31, although R31 is hardwired as all zeros. The register file 43 has four read ports and two write ports which allow the sourcing (sinking) of operands (results) to both the integer execution datapath and the address unit 19. A bus structure 44 connects two of the read ports of the register file 43 to the selected inputs of the ALU 40, the shifter 41 or the multiplier 42 as specified by the control bits of the decoded instruction on lines 28 from the instruction unit 18, and connects the output of the appropriate function to one of the write ports of the register file to store the result. That is, the address fields from the instruction are applied by the lines 26 to select the registers to be used in executing the instruction, and the control bits 28 define the operation in the ALU, etc., and define which internal busses of the bus structure 44 are to be used when, etc.

Detailed Description Text (9):
The A-box or address unit 19 is shown in more detail in FIG. 4. The A-box 19 includes five functions: address translation using a datapath translation buffer 48, a load silo 49 for incoming data, a write buffer 50 for outgoing write data, an interface 51 to a data cache, and the external interface 52 to the bus 11. The address translation datapath has the displacement adder 53 which generates the effective address (by accessing the register file 43 via the second set of read and write ports, and the PC), and the data TB 48 which generates the physical address on address bus 54.

Detailed Description Text (10):
The datapath translation buffer 48 caches a number (e.g., thirty-two) of the recently-used data-stream page table entries (as described below) for pages of 8 Kbyte size. Each entry supports any of four granularity hint block sizes, and a detector 55 is responsive to the granularity hint as described in application Ser. No. 547,630 to change the number of low-order bits of the virtual address passed through from virtual address bus 56 to the physical address bus 54.

Detailed Description Text (11):
For load and store instructions, the effective 43-bit virtual address is presented to TB 48 via bus 56. If the PTE of the supplied virtual address is cached in the TB 48, the PFN and protection bits for the page which contains the address are used by the address unit 19 to complete the address translation and access checks.

Detailed Description Text (12):
The on-chip pipelined floating point unit 17 or F-box as shown in more detail in FIG. 5 is capable of executing both DEC and IEEE floating point instructions according to the instruction set described in application Ser. No. 547,630. The floating point unit 17 contains a 32-entry, 64-bit, floating point register file 61 which includes floating-point registers F0 to F31, and contains a floating point arithmetic and logic unit 62. Divides and multiplies are performed in a multiply/divide circuit 63. A bus structure 64 interconnects two read ports of the register file 61 to the appropriate functional circuit as directed by the control bits of the decoded instruction on lines 28 from the instruction unit 18. The registers selected for an operation are defined by the output bus 26 from the instruction decode. The floating point unit 17 can accept an instruction every cycle, with the exception of floating point divide instructions, which can be accepted only every several cycles. A latency of more than one cycle is exhibited for all floating point instructions, during which the integer unit can continue to

execute other instructions.

Detailed Description Text (13):
In an example embodiment, the CPU 10 has an 8 Kbyte data cache 59, and 8 Kbyte instruction cache 21, with the size of the caches depending on the available chip area. The on-chip data cache 59 is write-through, direct mapped, read-allocate physical cache and has 32-byte (1-hexaword) blocks. The system may keep the data cache 59 coherent with memory 12 by using an invalidate bus, not shown.

Detailed Description Text (14):
The instruction cache 21 may be 8 Kbytes, or 16 Kbytes, for example, or may be larger or smaller, depending upon die area. Although described above as using physical addressing with a TB 36, it may also be a virtual cache, in which case it will contain no provision for maintaining its coherence with memory 12. If the cache 21 is a physical addressed cache the chip will contain circuitry for maintaining its coherence with memory: (1) when the write buffer 50 entries are sent to the bus interface 52, the address will be compared against a duplicate instruction cache 21 tag, and the corresponding block of instruction cache 21 will be conditionally invalidated; (2) the invalidate bus will be connected to the instruction cache 21.

Detailed Description Text (15):
The main data paths and registers in the CPU 10 are all 64-bits wide. That is, each of the integer registers 43, as well as each of the floating point registers 61, is a 64-bit register, and the ALU 40 has two 64-bit inputs 40a and 40b and a 64-bit output 40c. The bus structure 44 in the execution unit 16, which actually consists of more than one bus, has 64-bit wide data paths for transferring operands between the integer registers 43 and the inputs and output of the ALU 40. The instruction decoder 23 produces register address outputs 26 which are applied to the addressing circuits of the integer registers 43 and/or floating point registers 61 to select which register operands are used as inputs to the ALU 41 or 62, and which of the registers 43 or registers 61 is the destination for the ALU (or other functional unit) output.

Detailed Description Text (16):
A feature of the CPU 10 of FIGS. 1-6 of this example embodiment is its RISC characteristic. The instructions executed by this CPU 10 are always of the same size, in this case 32-bits, instead of allowing variable-length instructions. The instructions execute on average in one machine cycle (pipelined as described below, and assuming no stalls), rather than a variable number of cycles. The instruction set includes only register-to-register arithmetic/logic type of operations, or register-to-memory (or memory-to-register) load/store type of operations, and there are no complex memory addressing modes such as indirect, etc. An instruction performing an operation in the ALU 40 always gets its operands from the register file 43 (or from a field of the instruction itself) and always writes the result to the register file 43; these operands are never obtained from memory and the result is never written to memory in the same instruction that performs the ALU operation. Loads from memory are always to a register in register files 43 or 61, and stores to memory are always from a register in the register files.

Detailed Description Text (17):
Referring to FIG. 6, the CPU 10 has a seven stage pipeline for integer operate and memory reference instructions. FIG. 6 is a pipeline diagram for the pipeline of execution unit 16, instruction unit 18 and address unit 19. The floating point unit 17 defines a pipeline in parallel with that of the execution unit 16, but ordinarily employs more stages to execute. The seven stages are referred to as S0-S6, where a stage is to be executed in one machine cycle (clock cycle). The first four stages S0, S1, S2 and S3 are executed in the instruction unit 18, and the last three stages S4, S5 and S6 are executed in one or the other of the execution unit 16 or address unit 19, depending upon whether the instruction is an operate or a

load/store.

Detailed Description Text (18):
The first stage S0 of the pipeline is the instruction fetch or IF stage, during which the instruction unit 18 fetches new instructions from the instruction cache 21, using the PC 33 address as a base. The second stage S1 is the evaluate stage, during which two fetched instructions are evaluated to see if dual issue is possible. The third stage S2 is the decode stage, during which the instructions are decoded in the decoder 23 to produce the control signals 28 and register addresses on lines 26. The fourth stage S3 is the register file 43 access stage for operate instructions, and the instruction issue stage. The fifth stage S4 is cycle-one of the computation (in ALU 40, for example) if it is an operate instruction, and also the instruction unit 18 computes the new PC 33 in address generator 29; if it is a memory reference instruction the address unit 19 calculates the effective data stream address using the adder 53. The sixth stage S5 is cycle-two of the computation (e.g., in ALU 40) if it is an operate instruction, and also the data TB 48 lookup stage for memory references. The last stage S6 is the write stage for operate instructions having a register write, during which, for example, the output 40c of the ALU 40 is written to the register file 43 via the write port, and is the data cache 59 or instruction cache 21 hit/miss decision point for instruction stream or data stream references.

Detailed Description Text (19):
Referring to FIG. 7, the formats of the various types of instructions of the instruction set executed by the CPU 10 of FIGS. 1-7 are illustrated. Using the instruction formats of FIG. 7, the CPU of FIG. 1 executes an instruction set which includes nine types of instructions. These include (1) integer load and store instructions, (2) integer control instructions, (3) integer arithmetic, (4) logical and shift instructions, (5) byte manipulation, (6) floating point load and store, (7) floating point control, (8) floating point arithmetic, and (9) miscellaneous. The instruction set is described in more detail in application Ser. No. 547,630.

Detailed Description Text (20):
One type is a memory (i.e., load and store) instruction 70, which contains a 6-bit opcode in bits <31:26>, two 5-bit register address fields Ra and Rb in bits <25:21> and <20:16>, and a 16-bit signed displacement in bits <15:0>. This instruction is used to transfer data between registers 43 and memory (memory 12 or caches 59 or 20), to load an effective address to a register of the register file 43, and for subroutine jumps. The displacement field <15:0> is a byte offset; it is sign-extended and added to the contents of register Rb to form a virtual address. The virtual address is used as a memory load/store address or a result value depending upon the specific instruction.

Detailed Description Text (21):
The branch instruction format 71 is also shown in FIG. 7, and includes a 6-bit opcode in bits <31:26>, a 5-bit address field in bits <25:21>, and a 21-bit signed branch displacement in bits <20:0>. The displacement is treated as a longword offset, meaning that it is shifted left two bits (to address a longword boundary), sign-extended to 64-bits and added to the updated contents of PC 33 to form the target virtual address (overflow is ignored).

Detailed Description Text (22):
The operate instructions 72 and 73 are of the formats shown in FIG. 7, one format 72 for three register operands and one format 73 for two register operands and a literal. The operate format is used for instructions that perform integer register operations, allowing two source operands and one destination operand in register file 43. One of the source operands can be a literal constant. Bit-12 defines whether the operate instruction is for a two source register operation or one source register and a literal. In addition to the 6-bit opcode at bits <31:26>, the operate format has a 7-bit function field at bits <11:5> to allow a wider range of

choices for arithmetic and logical operation. The source register Ra is specified in either case at bits <25:21>, and the destination register Rc at <4:0>. If bit-12 is a zero, the source register Rb is defined at bits <20:16>, while if bit-12 is a one then an 8-bit zero-extended literal constant is formed by bits <20:13> of the instruction. This literal is interpreted as a positive integer in the range 0-255, and is zero-extended to 64-bits.

Detailed Description Text (23):
FIG. 7 also illustrates the floating point operate instruction format 74, used for instructions that perform floating point register 61 to floating point register 61 operations. The floating point operate instructions contain a 6-bit opcode at bits <31:26> as before, along with an 11-bit function field at bits <15:5>. There are three operand fields, Fa, Fb and Fc, each specifying either an integer or a floating-point operand as defined by the instruction; only the registers 61 are specified by Fa, Fb and Fc, but these registers can contain either integer or floating-point values. Literals are not supported. Floating point conversions use a subset of the floating point operate format 74 of FIG. 7 and perform register-to-register conversion operations; the Fb operand specifies the source and the Fa operand should be reg-31 (all zeros).

Detailed Description Text (24):
The other instruction format 75 of FIG. 7 is that for privileged architecture library (PAL or PALcode) instructions, which are used to specify extended processor functions. In these instructions a 6-bit opcode is present at bits <31:26> as before, and a 26-bit PALcode function field <25:0> specifies the operation. The source and destination operands for PALcode instructions are supplied in fixed registers that are specified in the individual instruction definitions. A PALcode instruction usually uses a number of instructions of formats 70-74 stored in memory to make up a more complex instruction which is executed in a privileged mode, as part of the operating system, for example.

Detailed Description Text (25):
The six-bit opcode field <31:26> in the instruction formats of FIG. 7 allows only 2.sup.6 or sixty-four different instructions to be coded. Thus the instruction set would be limited to sixty-four. However, the "function" fields in the instruction formats 72, 73 and 74 allow variations of instructions having the same opcode in bits <31:26>. Also, the "hint" bits in the jump instruction allow variations such as JSR or RET.

Detailed Description Text (26):
Referring to FIG. 8, the format 76 of the virtual address asserted on the internal address bus 56 is shown. This address is nominally 64-bits in width, but of course practical implementations at present use much smaller addresses. For example, an address of 43-bits provides an addressing range of 8-Terabytes. The format includes a byte offset 77 of, for example, 13-bits to 16-bits in size, depending upon the page size employed. If pages are 8-Kbytes, the byte-within-page field 77 is 13-bits, while for 16-Kbyte pages the field 77 is 14-bits. The format 76 as shown includes three segment fields 78, 79 and 80, labelled Seg1, Seg2 and Seg3, also of variable size depending upon the implementation. The segments Seg1, Seg2, and Seg3 can be 10-to-13 bits, for example. If each segment size is 10-bits, then a segment defined by Seg3 is 1K pages in length, a segment for Seg2 is 1M pages, and a segment for Seg1 is 1 G pages. The page frame number (PFN) field in the PTE is always 32-bits wide; thus, as the page size grows the virtual and physical address size also grows.

Detailed Description Text (27):
The physical addresses are at most 48-bits, but a processor may implement a smaller physical address space by not implementing some number of high-order bits. The two most significant implemented physical address bits select a caching policy or implementation-dependent type of address space. Different implementations may put

different uses and restrictions on these bits as appropriate for the system. For example, in a workstation with a 30-bit <29:0> physical address space, bit <29> may select between memory and I/O, and bit <28> may enable or disenable caching in I/O space and must be zero in memory space.

Detailed Description Text (28):
Typically, in a multitasking system, several processes may reside in physical memory 12 (or caches) at the same time, so memory protection and multiple address spaces (using address space numbers) are used by the CPU 10 to ensure that one process will not interfere with either other processes or the operating system. To further improve software reliability, four hierarchical access modes (privilege modes) provide memory access control. They are, from most to least privileged: kernel, executive, supervisor, and user, referring to operating system modes and application programs. Protection is specified at the individual page level, where a page may be inaccessible, read-only, or read/write for each of the four access modes. Accessible pages can be restricted to have only data or instruction access.

Detailed Description Text (30):
The translation buffers 36 and 48 store a number of the page table entries 81, each associated with a tag consisting of certain high-order bits of the virtual address to which this PFN is assigned by the operating system. For example, the tag may consist of the fields 78, 79 and 80 for the Seg1, Seg2 and Seg3 values of the virtual address 76 of FIG. 8. In addition, each entry contains a valid bit, indicating whether or not the entry has been invalidated, as when the TB is flushed. It is conventional to flush the TB when a context switch is made, invalidating all the entries; the features of the invention, however, allow continued use of entries still useful, so performance is improved. To this end, the translation buffers 36 and 48 include in addition an address space number field, perhaps sixteen bits in width, loaded from the process control block as will be described.

Detailed Description Text (31):
Referring to FIG. 10, the virtual address 76 on the bus 56 (seen in FIG. 8) is used to search for tag match for a PTE in the translation buffer, and, if not found, then Seg1 field 78 is used to index into a first page table 85 found at a base address stored in an internal register 86 referred to as the page table base register. The entry 87 found at the Seg1 index in table 85 is the base address for a second page table 88, for which the Seg2 field 79 is used to index to an entry 89. The entry 89 points to the base of a third page table 90, and Seg3 field 80 is used to index to a PTE 91. The physical page frame number from PTE 91 is combined with the byte offset 77 from the virtual address, in adder 92, to produce the physical address on bus 54. As mentioned above, the size of the page mapped by a PTE, along with size of the byte offset 77, can vary depending upon the granularity hint.

Detailed Description Text (32):
According to the invention, in addition to matching the tag field of the virtual address 76 with the tag field 93 of the translation buffer 36 or 48, an address space number 94 stored in the translation buffer is compared to the address space number in the current state of the CPU 10, stored in an ASN register 95 which is one of the internal processor registers. If the address space match field (bit <4>, Table A) is clear (zero), the current ASN and the field 94 must match for the PTE to be used, but if set (logic one) then there need not be a match, i.e., the address space numbers are ignored. A particular feature of the invention, however, is an additional match-disable bit 96 stored for each PTE, disabling the address space match feature under certain conditions (i.e., when the virtual machine monitor process is being executed).

Detailed Description Text (33):
The match-disable bit is not required to be maintained on each entry in the

translation buffer. Whether or not address-space matches should be disabled is properly a function of the execution environment of the CPU rather than of the virtual address. When the virtual machine monitor is being executed (as discussed below), address-space matches are disabled; when a virtual machine or some process in a VM is being executed, address-space matches are enabled. In another embodiment of the invention, the match-disable bit could be stored globally in the translation buffer. In yet another embodiment of the invention, the match-disable bit could be maintained in the CPU itself. In either case, its value would be changed on transition from the VMM to a VM or from a VM to the VMM, and its current value must be made available to the match comparison logic in the translation buffer.

Detailed Description Text (34):
The CPU 10 generates memory references by first forming a virtual address 76 on bus 56, representing the address within the entire virtual range 97 as seen in FIG. 11, defined by the 43-bit address width referred to, or that portion of the address width used by the operating system. Then using the page tables 85, 88, 90 in memory, or the translation buffer 36 or 48, the virtual address is translated to a physical address represented by an address map 98; the physical memory is constrained by the size of the main memory 12. The translation is done for each page (e.g., an 8 Kbyte block), so a virtual page address for a page 99 in the virtual memory map 97 is translated to a physical address 99' for a page (referred to as a page frame) in the physical memory map 98. The page tables are maintained in memory 12 or cache 20 to provide the translation between virtual address and physical address, and the translation buffer is included in the CPU to hold the most recently used translations so a reference to the page tables in memory 12 need not be made in most cases to obtain the translation before a data reference can be made; the time needed to make the reference to a page table in memory 12 would far exceed the time needed to obtain the translation from the translation buffer. Only the pages used by tasks currently executing (and the operating system itself) are likely to be in the physical memory 12 at a given time; a translation to an address 99' is in the page table 85, 88, 90 for only those pages actually present in physical memory 12. When the page being referenced by the CPU 10 is found not to be in the physical memory 12, a page fault is executed to initiate a swap operation in which a page from the physical memory 12 is swapped with the desired page maintained in the disk memory 13, this swap being under control of the operating system. Some pages in physical memory 12 used by the operating system kernel, for example, or the page tables 85, 88, 90 themselves, are in fixed positions and may not be swapped to disk 13 or moved to other page translations; most pages used by executing tasks, however, may be moved freely within the physical memory 12 by merely keeping the page tables updated.

Detailed Description Text (35):
A process (or task) is a basic entity that is scheduled for execution by the CPU 10. A process represents a single thread of execution and consists of an address space and both hardware and software context. The hardware context is defined by the integer registers 43 and floating point registers 61, the processor status contained in internal processor registers, the program counter 33, four stack pointers, the page table base register 86, the address space number 95, and other values depending upon the CPU design. The software context of a process is defined by operating system software and is system dependent. A process may share the same address space with other processes or have an address space of its own; there is, however, no separate address space for system software, and therefore the operating system must be mapped into the address space of each process. In order for a process to execute, its hardware context must be loaded into the integer registers 43, floating point registers 61, and internal processor registers. While a process is executing, its hardware context is continuously updated, as the various registers are loaded and written over. When a process is not being executed, its hardware context is stored in memory 12. Saving the hardware context of the current process in memory, followed by loading the hardware context for a new process, is referred to as context switching. Context switching occurs as one process after

another is scheduled by the operating system for execution. The hardware context of a process is defined by a privileged part and nonprivileged part. The privileged part is stored in memory in a 128-byte block 100 as shown in FIG. 12 when a process is not executing, and context is switched by a privileged instruction. There is one block 100 for each process. The nonprivileged part is context switched by the operating system software.

Detailed Description Text (36):
Referring to FIG. 12, the context block 100 contains four stack pointers in fields 101, these being stack pointers for the kernel, the executive, the supervisor and the user. The page table base register 86 is in field 102. The address space number for this process (to be loaded to register 95) is in field 103. Other fields 104 are for values not material here. The location of this block 100 in memory is specified for the current process by a context block base register 105. A swap context instruction saves the privileged context of the current process into the context block specified by this register 105, loads a new value into the register 105, and then loads the privileged context of the new process from the new block 100 into the appropriate hardware registers 43, etc.

Detailed Description Text (37):
The architecture as described above allows a processor to implement address space numbers (process tags) to reduce the need for invalidation of cached address translations in the translation buffer for process-specific addresses when a context switch occurs. The address space number for the current process is loaded by a privileged instruction from field 103 into an internal processor register 95.

Detailed Description Text (38):
In the page table entry 81 of FIG. 9 and Table A, there is a field (bit <4>) called "address space match." This feature allows an operating system to designate locations in the system's virtual address space 97 which are shared among all processes. Such a virtual address refers to the same physical address in each process's address space.

Detailed Description Text (39):
The CPU 10 of FIGS. 1-5 may employ a "virtual machine system" which uses a combination of hardware, firmware, and software mechanisms to create the illusion of multiple, independent simulated or virtual computers each running on the same physical computer. Virtual machine systems create a set of virtual machines (VMs) in an analogous fashion to how time sharing systems create a set of independent user processes. In virtualizing the architecture of FIG. 1-5, it is desirable from performance and functional standpoints to provide the address-space-match feature to virtual machines through hardware means.

Detailed Description Text (40):
Virtual machines are created by a layer executing on the CPU called the virtual machine monitor (VMM). The VMM is in control of the hardware, including memory management (address translation and protection mechanisms) and the page tables. Each virtual machine runs in a separate virtual address space in the range 97, and has a distinct set of address space numbers assigned by the VMM. The VMM also runs in its own, independent set of virtual address spaces in the range 97.

Detailed Description Text (41):
The purpose of the invention is to maximize system performance, while allowing the virtual machines to run in kernel mode so they can execute privileged instructions (otherwise they would have to use traps for these operations, at considerable performance penalty); to do this the VMM must constrain the VMs. The problem addressed in implementing this invention is to keep the address spaces of the several VMs and the VMM itself separate from each other, while at the same time maximizing system performance by providing the match function in the TB to the VMs. A further constraint on the solution is that it is expected that the VMs and the

VMM will use the same virtual addresses for different purposes. Thus, it is not a solution to allocate separate address regions to the VMM from those allocated to the VMs.

Detailed Description Text (42):
To describe this invention, an example is illustrative. Suppose that the hardware provides fifteen address spaces for use by software, numbered ASN-0 through ASN-14. Assume that address spaces ASN-0 and ASN-9 are dedicated for use by the VMM. On this example system, there are running two virtual machines, A and B, each of which is running five user processes. One possible assignment of address spaces to this mix would be to dedicate address spaces ASN-1 through ASN-5 to VM A and address spaces ASN-6, -7, -8, -10, and -11 to VM B. This example is illustrated in FIG. 13.

Detailed Description Text (43):
To preserve system security and integrity, it is required that the two virtual machines be completely independent. That means that they cannot reference each other's memory. Similarly, the VMM's memory must also be isolated from the virtual machines. However, within a virtual machine, the address-space-match feature should work correctly, allowing the individual processes to share memory, under control of each VM's operating system.

Detailed Description Text (44):
A straightforward translation buffer design assigns each address space an address space number (ASN). Part of the CPU's state information is the ASN assigned to the currently running entity or process, register 95. TB entries are tagged in field 94 with the ASN of the address space to which they belong. In addition, there is a bit or field in the TB (Table A, ASM bit <4>) which indicates that the entry matches any address space, in effect overriding the ASN tag 94.

Detailed Description Text (46):
To minimize the TB flushes, and thus improve overall system performance, some restriction is imposed on the VMM and the TB construction is changed, according to the invention. The restriction imposed on the VMM is to require it to use the disable match feature. This reserves the address space match feature for use by the virtual machines, and guarantees that no VMM address will be mapped by the TB with the match field set.

Detailed Description Text (47):
The TB itself is modified by adding another piece of CPU state called "disable match" which is the field 96 of FIG. 10. The VMM determines the value of the disable match field 96 on a per-address-space basis and forwards the current value to the TB. The field 96 is a flag that can be either set or clear. If the disable match field 96 is clear, the TB will match a reference if the address is correct and (1) the ASN in the TB entry matches the CPU's current ASN, or (2) the match field in the TB entry is set.

Detailed Description Text (49):
where ADDR.TAG is the tag fields 78, 79, 80 of the virtual address 76, TB.TAG is the tag 93 in the translation buffer, CPU.ASN is the value stored in the processor register 95 from the field 103 of the control block, TB.ASN is the address space number stored in the field 96 of the translation buffer, TB.ASM is the match bit, and TB.DIS is the diable bit in field 96 of the translation buffer. These values are applied to a logic circuit 106 in the A-box of FIG. 4 to generate a match signal to indicate whether or not the PTE selected in the TB is to be used.

Detailed Description Text (50):
The logic implemented in the circuit 106 and given in the preceding paragraph in equation form may also be expressed a truth table as follows (in each case assuming the address tags match):

h e b b g e e e f c e g e ge

Detailed Description Text (51):
If the disable match field 96 is set, the TB will match a reference only if the address is correct and the ASN 94 in the TB entry matches the CPU's current ASN 95. In effect, the disable match state bit 96 overrides the match field in the TB.

Detailed Description Text (53):
In another embodiment of the invention, a multi-bit field for a virtual machine number VMN is added to each translation buffer entry as seen in FIG. 15, instead of the single-bit disable indicator as discussed above. Likewise, a multi-bit virtual machine number VMN is added to the state of the CPU in an internal processor register, like the address space number field 95. The translation buffer then contains logic to match a virtual address on the tags and match the ASNs, as before, and also logic to match on the virtual machine numbers. The logic implemented may be expressed

Detailed Description Text (55):
In this embodiment, each VM is assigned one (or more) VMN, and the VMM is assigned one (or more) VMN, and each must maintain operation using a VMN distinct to itself. In contrast to the previous embodiment, however, the translation buffer does not need to be cleared upon any switch between VMs. An additional advantage is the virtual machine monitor can use the address space match to share translation buffer entries among its processes. The disadvantage of this embodiment is that more bits are needed in the translation buffer entries.

Detailed Description Paragraph Table (4):
TABLE A Page Table Entry Fields in the page table entry are interpreted as follows: Bits Description <0> Valid (V) - Indicates the validity of the PFN field. <1> Fault On Read (FOR) - When set, a Fault On Read exception occurs on an attempt to read any location in the page. <2> Fault On Write (FOW) - When set, a Fault On Write exception occurs on an attempt to write any location in the page. <3> Fault on Execute (FOE) - When set, a Fault On Execute exception occurs on an attempt to execute an instruction in the page. <4> Address Space Match (ASM) - When set, this PTE matches all Address Space Numbers. For a given VA, ASM must be set consistently in all processes. <6:5> Granularity hint (GH) - Software may set these bits to a non-zero value to supply a hint to the translation buffer that a block of pages can be treated as a single larger page. <7> Reserved for future use. <8> Kernel Read Enable (KRE) - This bit enables reads from kernel mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in kernel mode, an Access Violation occurs. This bit is valid even when V = 0. <9> Executive Read Enable (ERE) - This bit enables reads from executive mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in executive mode, an Access Violation occurs. This bit is valid even when V = 0. <10> Supervisor Read Enable (SRE) - This bit enables reads from supervisor mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in supervisor mode, an Access Violation occurs. This bit is valid even when V = 0. <11> User Read Enable (URE) - This bit enables reads from user mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in user mode, an Access Violation occurs. This bit is valid even when V = 0. <12> Kernel Write Enable (KWE) - This bit enables writes from kernel mode. If this bit is a 0 and a STORE is attempted while in kernel mode, an Access Violation occurs. This bit is valid even when V = 0. <13> Executive Write Enable (EWE) - The bit enables writes from executive mode. If this bit is a 0 and a STORE is attempted while in executive mode, an Access Violation occurs. <14> Supervisor Write Enable (SWE) - This bit enables writes from supervisor mode. If this bit is a 0 and a STORE is attempted while in executive mode, an Access Violation occurs. <15> User Write Enable (UWE) - This bit enables writes from user mode. If this bit is a 0 and a STORE is attempted while in user mode, Access Violation occurs. <31:16> Reserved for software. <63:32> Page Frame Number (PFN) - The PFN field always points to a page boundary. If V is set, the PFN is concatenated with the

Byte Within Page bits of the virtual address to obtain the physical address. If V is clear, this field may be used by software.

CLAIMS:

1. A method of operating a processor having a translation buffer for translating a virtual address to a physical address, said method comprising the steps of:

storing in said translation buffer a plurality of page table entries, each page table entry containing a page frame number indexed by a virtual address tag;

also storing in said translation buffer for each said page table entry an address space number, and storing in said translation buffer for each said page table entry an address space match entry; where said address space number is a value corresponding to a process executed on said processor, said match entry is a field having one value indicating that the address space number is to be required to be matched and having another value indicating that the address space number is not required to be matched;

storing a current number in said processor as part of a state of said processor;

and storing a third match value having one condition indicating that said match entry is to be disabled and having another condition indicating that said match entry is not to be disabled;

comparing said virtual address tag with a field of a virtual address generated by said processor, and also comparing said address space number with said current number, if comparing said virtual address tag with said field of said virtual address produces a match, and

if said step of comparing of said address space number and said current number produces said match, and said match entry is of said one value, then using said page frame number for a memory reference; and

if said match entry is said another value, then using said page frame number for said memory reference regardless of whether said address space number matches said current number, if said third match value is in said one condition.

5. The method of operating said processor according to claim 1 including the steps of fetching instructions from an external memory, decoding said instructions, and executing said instructions, said executing including accessing said external memory for read and write data; said external memory storing a page table of said page table entries.

8. A method of operating a processor system having a CPU and a memory, the CPU having a translation buffer for translating virtual addresses to physical addresses, said method of operating comprising the steps of:

storing in said translation buffer a number of page table entries, each page table entry containing a virtual address tag, a page frame number, and an address space number to characterize a location in said memory referenced by said page frame number;

also storing in said translation buffer for each page table entry (a) an address space match indication having one value indicating that said address space number must match a current value stored as part of a state of said processor and having another value indicating that said address space number need not match said current value, and (b) a match disable indication having a first value specifying that said address space match indicator is to be operable for said entry and having a second

value indicating that said address space match indicator is not to be operable for an entry;

comparing a field of a virtual address generated by said processor with a virtual address tag of one of said page table entries in said translation buffer, and

if said step of comparing indicates a match, and said address space match indication is of said another value, and said address space number matches a value stored as part of the state of said processor, then using said page frame number for addressing said memory;

if said step of comparing indicates said match, and said address space match indication is of said one value, regardless of whether said address space number matches said current value, using said page frame number for addressing said memory;

if said step of comparing indicates a match, and said match disable indication is of said first value, regardless of whether said address space match indication is of said one value or said another value, then using said page frame number for addressing said memory only if said address space number matches said current value.

9. The method of operating said processor system according to claim 8 including fetching instructions from an external memory, decoding said instructions, and executing said instructions, said executing including accessing said external memory for read and write data; said external memory storing a page table of said page table entries.

16. A processor comprising:

addressing means including a translation buffer for translating virtual addresses to physical addresses, said translation buffer storing a plurality of page table entries, each page table entry containing a page frame number indexed by a virtual address tag;

said translation buffer including means for storing for each said page table entry an address space number, means for storing an address space match entry, and means for storing a match disable indicator; where said address space number corresponds to a process executed on said processor, said match entry is a field having one value indicating that the address space number is to be required to be matched and having another value indicating that the address space number is not required to be matched, and said match disable indicator is an indication having a first value indicating that said match entry is to be ignored and having a second value indicating that said match entry is not to be ignored;

means for maintaining in said processor as part of a state of said processor a current number representing said address space number;

first means for comparing said virtual address tag with a field of a virtual address generated by said processor, and also second means for comparing said address space number with said current number, and

if both of said first and second means for comparing produce a match, and said match entry is of said one value, then said addressing means using said page frame number for a memory reference;

if said means for comparing said virtual address tag with said field of said virtual address produces said match, and if said match entry is of said another value, then said addressing means using said page frame number for said memory reference regardless of whether said second first means for comparing finds that

said address space number matches said current number, unless said disable indicator is set to said first value.

17. The processor according to claim 16 including means for fetching instructions from an external memory, decoding said instructions, and executing said instructions, said executing including accessing said external memory for read and write data; said external memory storing a page table of said page table entries.

18. The processor according to claim 17 including means for generating virtual addresses used for said fetching of instructions and said accessing said external memory for data, said virtual addresses being compared to address tags in said translation buffer.

20. A processor system having a CPU and a memory, said processor system comprising:

a) means in said CPU for fetching instructions from said memory, means in said CPU for decoding said instructions, and means in said CPU for executing said instructions, said means for executing including means for accessing said memory for read and write data;

b) means in said CPU for generating a virtual address used by said means for fetching of instructions and by said means for accessing said memory for data;

c) a page table stored in said memory and containing a plurality of page table entries, each page table entry including a page frame number referencing a different page of said memory;

d) means for translating said virtual address to a physical address for said memory, said means for translating including a translation buffer storing a number of said page table entries;

e) and means for addressing said memory using the page frame number from said translation buffer and using a part of said virtual address;

f) said translation buffer storing for each said page table entry an address tag and an address space number, and storing an address space match entry, where said address space number is a value corresponding to a process executed on said CPU, and said match entry is an indicator having one value indicating that the address space number is to be required to be matched and having another value indicating that the address space number is not required to be matched; said translation buffer also storing for each said page table entry a match disable indicator; means in said CPU for storing a current number representing an address space value maintained as part of a state of said CPU;

g) compare means in said translation buffer for first comparing said address tag with a field of said virtual address generated by said CPU, and also second comparing said address space number with said current number maintained as part of the state of said CPU, and

if both of said first and second comparing by said compare means produce a match, and said match entry is of said one value, then said addressing means using said page frame number for a memory reference; or

if comparing said address tag with said field of said virtual address produces said match, and if said match entry is of said another value, then said addressing means using said page frame number for said memory reference regardless of whether said address space number matches said current number, unless said disable match indicator is set.

h e b b g e e e f c e g e ge